Sistemas Informáticos
Curso 2002-03

ENTORNO DE PRUEBA DE ALGORITMOS DE BIN-PACKING TRIDIMENSIONAL

Dianderas La Torre, Zadith
López Teso, Fernando
Rodríguez de Pastors, Javier

Dirigido por:

Julio Septién del Castillo
Departamento de Arquitectura de Computadores y Automática

Facultad de Informática
Universidad Complutense de Madrid

ÍNDICE

1. RESUMEN: ENTORNO DE PRUEBA DE ALGORITMOS DE BIN-PACKING TRIDIMENSIONAL

Descripción:

Se ha diseñado un entorno que permite la aplicación de diversos algoritmos de bin-packing 3-D (empaquetamiento en tres dimensiones, como los bultos que se almacenan en un contenedor) para un conjunto de cajas de tamaños diversos que se ubican en un volumen dado, con algunas restricciones (por ejemplo, sólo cajas en forma de prismas rectangulares).

Se da soporte gráfico a los siguientes aspectos:

a. Algoritmos de empaquetamiento tanto on-line (que ubican cada nueva caja a medida que van llegando, de un modo rápido y lo más eficiente posible) como off-line (que analizan el mejor empaquetado para un conjunto de cajas previamente conocido) , siempre teniendo como objetivo minimizar la altura del empaquetamiento.
b. Se incluiye también una herramienta de visualización tridimensional que permite representar gráficamente el empaquetado y su evolución a medida que avanza el empaquetamiento.
c. Se incluiyen herramientas que permitan comparar los resultados de los distintos algoritmos.

El proyecto ha sido realizado en un lenguaje de alto nivel como es C++ y para manejar los gráficos OpenGL.

La mayor parte de la pantalla está dedicada a la visualización de la simulación, representando una imagen de la FPGA y las tareas hardware que en ella han sido distribuidas por el algoritmo elegido, pudiendo el usuario escoger el tipo de vista, la posición de la cámara, la escala, emplear un zoom o incluso rotar los objetos de la imagen.

La aplicación, diseñada para un entorno Windows, necesita de las DLLs BORLNDMM.DLL y CC3250MT.DLL incluidas en el CD de instalación.

La resolución de la pantalla ha de ser 1024x768 o superior.

Los requisitos hardware de la aplicación incluyen un procesador a 400 MHz o superior y un mínimo 64 Mb de RAM.

2. SUMMARY: A BIN PACKING 3D ALGORITHM PROOF ENVIRONMENT

Description:

We have designed an environment that allows running multiple bin packing 3D algorithms (packing in a three dimensional space, like the packages that are stored in a container) that work with a set of boxes of different sizes that are placed in a certain volume, with estrictions (for example, the boxes have to be rectangular prism shaped).

There is graphics support for the following feaures:

a. Bin packing 3D algorithms, both online (that places the boxes on the fly, in a fast manner while trying to be as much efficient as posible) and offline (that search for the best placement of a previously known set of boxes), always trying to minimize the height component.
b. There is a 3D visualization tool that shows graphicaly the result of the bin packing 3D algorithms and it´s evolution while that algorithms are running.
c. There are tools that allows to compare the results of the several algorithms.

This proyect has been coded in a high level programming language (C++) and the graphics have been done with the OpenGL graphics library.

The greater part of the screen is used to show the simulation as an image that represents the FPGA and the RFU operations that have been placed on it by the choosed algorithm. User can choose the type of view, as well as the camera position, the scale, the use of the zoom and it's allowed to rotate the image objects.

This application, that runs under Windows, need the following DLLs: BORLNDMM.DLL and CC3250MT.DLL. These DLLs are included in the installation CD.

The screen resolution must be of 1024x768 or higher.

The hardware requirements of the application include a 400 MHz processor or higher and 64 Mb of RAM.

3. INTRODUCCIÓN: COLOCACIÓN RÁPIDA DE TAREAS PARA SISTEMAS DE COMPUTACIÓN RECONFIGURABLES

Como las FPGAs se vuelven cada vez más grandes y rápidas, tanto el número como la complejidad de los módulos que se pueden cargar en ellas aumenta, por lo que se pueden conseguir mejoras de rendimiento considerables explotando las FPGAs en sistemas hardware.

Además, tienen la capacidad de reconfigurar el chip "en caliente", con lo que se posibilita la implementación de sistemas hardware dinámicamente reconfigurables que se adaptan a la aplicación para mejorar el rendimiento de la misma. Las tareas hardware se pueden programar mientras la aplicación está ejecutándose, variando las configuraciones en distintas fases de la misma.

Desgraciadamente, se producen retardos relativamente largos al reprogramar las tareas hardware, haciendo que no se consigan mejoras de rendimiento realmente impresionantes en lo que a computación de propósito general se refiere.

Se han propuesto una serie de técnicas para paliar parcialmente los retardos al reconfigurar las tareas hardware. Algunas de éstas técnicas son:

- Técnicas en tiempo de compilación, como el prefetching y la compresión de configuraciones.
- Políticas de caché, que mantienen las operaciones más usadas, eliminando la necesidad de reprogramar al llamar a ésas operaciones.

A pesar de que éstos algoritmos son necesarios para un sistema reconfigurable práctico, todavía se necesitan herramientas CAD de diseño, rápidas y potentes, para llevar a cabo la administración de configuraciones de tareas hardware, tanto en modo on-line como en modo off-line.

En la versión off-line, el flujo de ejecución del programa se conoce a priori, con lo que el planificador y el componente de administración de configuraciones pueden hacer varias optimizaciones de las tareas hardware antes de que el sistema empiece a correr.

Por el contrario, en la versión on-line, la decisión de qué operaciones deben ser lanzadas no es conocida de antemano. El flujo de ejecución del programa no es conocido a priori, con lo que la administración de las configuraciones de las tareas hardware debe hacerse sobre la marcha.

Ambas versiones (on-line y off-line) de los algoritmos son importantes para los sistemas de computación reconfigurables.

La versión on-line es importante porque es muy difícil predecir el comportamiento en tiempo de ejecución de un programa general en tiempo de

compilación, por lo que se necesitan métodos de colocación on-line para algunas partes del los kernels de administración de la FPGA.

Los algoritmos off-line se pueden explotar para generar ubicaciones compactas para un grupo de tareas hardware, que serán ejecutadas en secuencia, es decir, como parte del código en un solo bloque básico. Además, las colocaciones generadas por las versiones off-line pueden servir como punto de partida a la hora de buscar soluciones para la versión on-line, ayudando a obtener mejores algoritmos de dicha clase. Por ello, la cualidad más importante de los algoritmos off-line es la calidad de las ubicaciones que genera, aunque sea un método muy lento.

La meta para la versión on-line es conseguir métodos eficientes a la hora de ubicar tareas hardware en el chip de forma rápida, de forma que puedan ser usados en un sistema de computación reconfigurable. Además de ser rápidos, dichos métodos también deben ser capaces de empaquetar de forma compacta los módulos en la FPGA, usando el área del chip de forma eficiente.

En el caso off-line, la meta es encontrar métodos para ubicar las tareas hardware en el chip de la forma tan compacta como sea posible.

Nuestro modelo de un sistema de computación reconfigurable.

Es un entorno en el que el sistema escoge en tiempo de ejecución entre dos implementaciones (hardware y software) de la misma función, basándose en perfiles de datos u otros criterios.

Una tarea hardware ri puede ser rechazada o aceptada en función de la capacidad de llevarla a cabo, teniendo en cuenta el estado actual de la FPGA. Si es rechazada, ésa misma función deberá ser realizada por la CPU, incurriendo en una penalización en el tiempo de ejecución. Usaremos el conjunto ACC para representar qué tareas hardware son aceptadas.

En nuestro modelo, se asume que no hay comunicación entre las tareas hardware, así como que han sido planificadas en tiempo de compilación. No se consideran las técnicas de caché sobre los módulos durante el tiempo de ejecución.

El conjunto RFUOPS = { r1, r2, ..., rN | ri = (wi, hi, si, ei)}, representa todas las tareas hardware definidas en el sistema, donde wi, hi, si y ei son todos enteros positivos con la restricción adicional de que ei debe ser mayor estrictamente que si. wi y hi son, respectivamente, el ancho y el largo de la implementación de la tarea hardware ri en la librería. si es el tiempo en el que la tarea hardware ri es invocada y ei – si es el tiempo que dicha tarea está residente en el sistema.

El motor de ubicación de tareas se puede invocar de dos formas: insertar un módulo que no haya sido insertado previamente o bien eliminar uno que sí que lo haya sido. En el caso de que hubiera un administrador de caché, las peticiones de inserción o eliminación le llegarían al motor de ubicación sólo cuando dichas operaciones debieran tener lugar realmente.

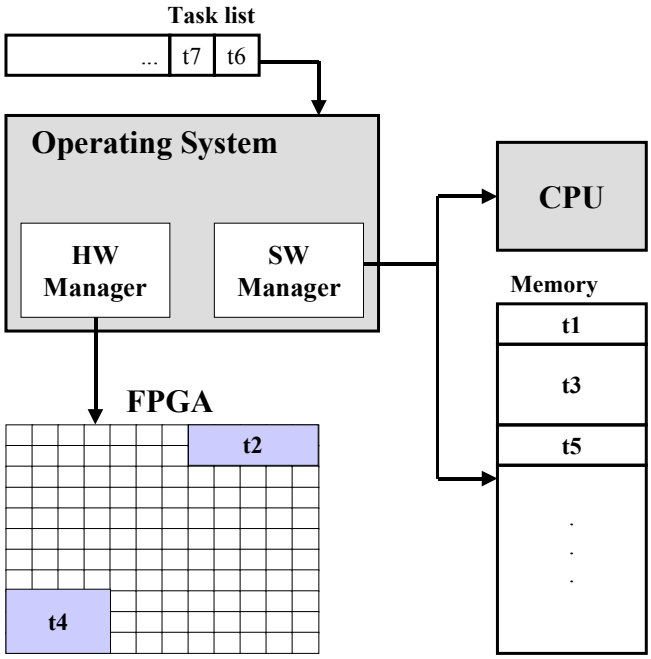


Figura 2

Se bloquearán las peticiones de inserción de tareas que no hayan mostrado mejoras de rendimiento.

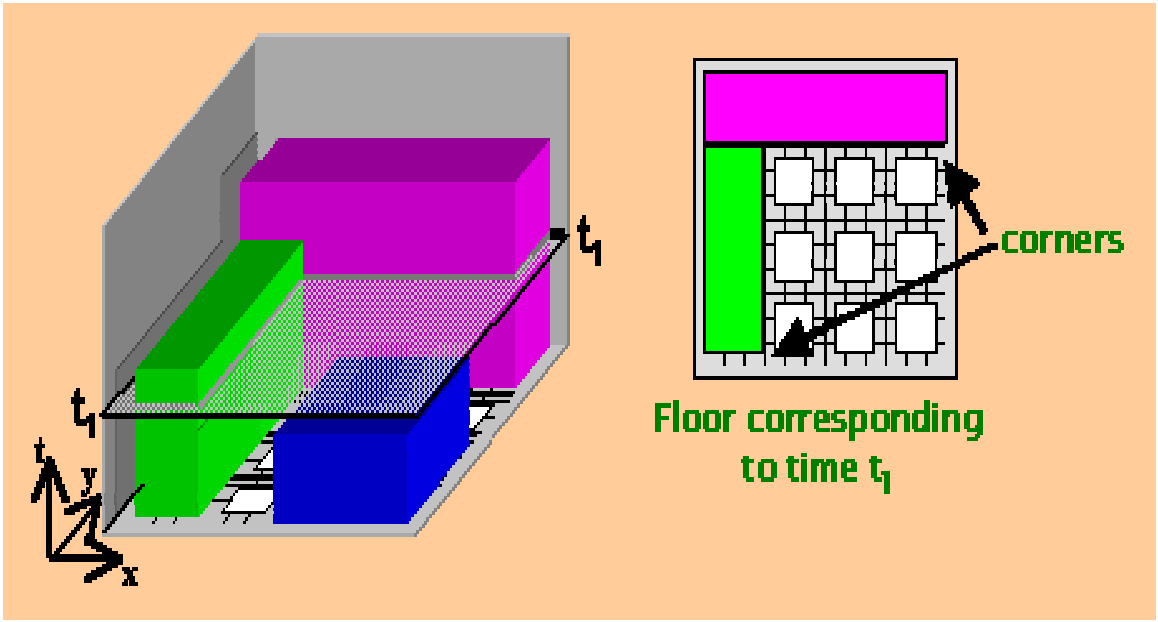
El conjunto ACC representa todas las tareas hardware que son aceptadas, además de sus posiciones en el chip. Dado un conjunto RFUOPS y unas dimensiones de la FPGA (W y H), el motor de ubicación de tareas decide donde colocarlas en el chip.

ACC = {(ri, xi, yi) | ri es una tarea hardware}
(xi, yi) son las coordenadas de la FPGA donde la tarea hardware ri será ubicada.

Deben cumplirse las siguientes condiciones:

xi >= 0, xi + wi < W
yi >= 0, yi + hi < W
W >= wi, ∀i = 1..N
H >= hi, ∀i = 1..N

La ubicación de tareas hardware en la FPGA se puede modelar como un problema de colocación de cajas en tres dimensiones. Tenemos una caja cuya base es un rectángulo con las mismas dimensiones que la FPGA (W * H), mientras que su altura representa el eje del tiempo. Por otro lado, las tareas hardware también se representan como cajas en tres dimensiones, cuya base es un rectángulo de wi * hi y cuya altura es el tiempo de ejecución, de forma que los extremos de la diagonal de la caja ri tienen coordenadas (xi, yi, si) y (xi + wi – 1, yi + hi – 1, ei – 1). A continuación se muestra un ejemplo:


t1
t1
t
y
x
corners
Floor corresponding
to time t1

4. LA APLICACIÓN

En este apartado se va a describir el programa desde el punto de vista técnico, comentando y explicando los métodos y clases más relevantes de cara a una mejor comprensión del proyecto y su implementación:

La aplicación consta de un interfaz gráfico de usuario al más puro estilo Windows generado a partir de Borland C++, se ha escogido este compilador por las facilidades que ofrece de cara a crear y gestionar utilidades de interacción con el usuario. En todo momento se ha buscado la comodidad de cara al usuario de manera que los menús propuestos para las distintas funcionalidades del programa sean lo más intuitivos posible. Podemos dividir la ventana principal del programa en dos: la parte encargada de recibir las órdenes y la que visualiza el resultado de las mismas; esta visualización consta a su vez de una parte textual que indicará la configuración actual del programa así como los mensajes generados por el propio algoritmo, además de la visualización propiamente dicha de la FPGA y las tareas que le llegan colocadas tras ser ubicadas. Prácticamente la totalidad de la pantalla es ocupada por esta representación gráfica del resultado del algoritmo, implementada con OpenGL.

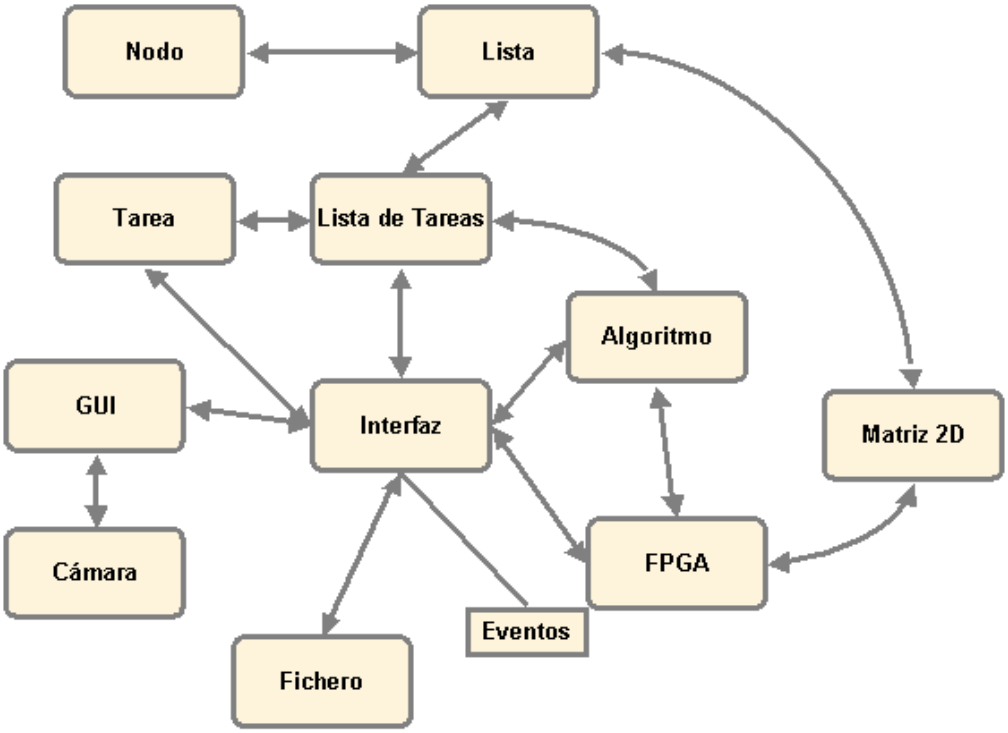
Los menús y botones ofertados al usuario completan la ventana, posibilitando la elección de las distintas funcionalidades tanto por teclado como desde ratón, entre dichas funcionalidades destacan las siguientes:

- Generar y guardar un archivo histórico.
- Cargar desde fichero un archivo histórico.
- Crear una nueva FPGA
- Cargar desde fichero varias FPGAs para su posterior elección
- Generar una lista de tareas hardware limitada por los parámetros elegidos por el usuario para su posterior tratamiento por el algoritmo de Bin Packing.
- Cargar desde fichero una lista sin procesar
- Modificación de las vistas y control de la reproducción de la simulación.
- Representación gráfica de los resultados y del código o mensajes generados por el algoritmo

El hecho de haber implementado la aplicación bajo Borland limita su ejecución a un entorno Windows debido a las librerías de ventanas que emplea, como hacen la totalidad de compiladores para C++, sin embargo, se ha buscado una fácil migración a Unix, separando claramente el interfaz gráfico de usuario del resto de funcionalidades, incluidas las visualizaciones de la FPGA y la lista de tareas. Con tan sólo crear un nuevo entorno de ventana válido, nuestra aplicación correría en Unix gracias a que el resto de la implementación ha sido realizada en C++ “puro” y el ya mencionado OpenGL.

La clase UnitPrincipal, encargada de la comunicación con el usuario está íntimamente relacionada con la clase Interfaz, siendo está última la encargada de llevar el peso de la aplicación: cada posibilidad que ofertan las distintas ventanas de UnitPrincipal se ve reflejada en un método de Interfaz, quien gestiona la comunicación entre las distintas clases del proyecto:

Diagrama de clases:



Encarguémonos ahora detenidamente de las clases y funcionalidades más relevantes.

UnitPrincipal

Es la clase encargada de la comunicación con el usuario, el GUI (graphic user interfaz). Prácticamente la totalidad de los métodos se crean automáticamente por Borland al escoger los objetos que van a ser utilizados, así como los métodos correspondientes a los eventos que se van a permitir, tales como pueden ser clicks de ratón o pulsaciones de teclado.

Se ha jugado con la propiedad Enabled de los objetos para restringir el acceso a ciertas acciones antes de cumplir los requisitos necesarios para que dichas acciones puedan realizarse con éxito, como por ejemplo el menú y el botón de grabación de histórico no se habilita hasta que el algoritmo no ha "empaquetado" la lista de tareas.

Otra propiedad que se ha aprovechado en cada objeto es Caption; la visualización de un texto en una componente puede implicar un acceso rápido a dicha componente desde teclado si empleamos el metacaracter & delante de la letra a la que se desea asociar el evento. De este modo conseguimos un desplazamiento por los menús

desde teclado, sin necesidad de emplear el ratón, ni tan siquiera para controlar la reproducción, dejando a la elección del usuario el modo en que interactuará con el sistema. Íntimamente relacionado con el control desde teclado está la propiedad TabOrder, que permite asociar un orden de tabulación a cada elemento que forma parte de la ventana: en cada uno de los formularios que aparecen se ha cuidado ofrecer un orden lógico de manera que la introducción de datos sea lo más cómoda posible, pero quizá esto se explicará mejor en el manual de usuario.

Tan sólo queda destacar la propiedad Hint, que muestra un texto cuando el ratón se posa en el objeto indicado. Esta propiedad resulta muy útil para añadir información adicional sobre el componente a usar de una manera discreta.

La principal complicación encontrada a la hora de implementar el interfaz gráfico de usuario ha sido crear las opciones en ejecución. Borland ofrece muchas comodidades para insertar y manipular objetos que se conocen en tiempo de compilación, sin embargo la gestión de componentes creados en ejecución se complica. Pese a esta ligera dificultad añadida, se ha preferido implementarlo de esta manera con el fin de conseguir mayor comodidad de cara al usuario, incluso al avanzado (el que desee modificar el código). De este modo, se emplean contenedores donde se colocan objetos de tipo TRadioButton (casillas de verificación agrupadas donde solamente una de ellas puede estar activada) para cada elección del usuario sobre un posible rango de valores. La principal causa de está elección es permitir una fácil incorporación de algoritmos en la clase Algoritmo. Al cargar el panel donde se muestran todos los algoritmos que el usuario puede escoger para manipular la lista de tareas, no se genera directamente, si no que se pregunta a la Interfaz cuántos algoritmos hay, se crean tantos TRadioButtons como corresponda y se les asigna el nombre que tiene cada uno de ellos asociado. El algoritmo elegido se escoge en este contenedor (ver Algoritmo).

Interfaz

Interfaz es la clase principal de la aplicación, por ella pasa prácticamente la totalidad del flujo de datos relevantes para el correcto funcionamiento del programa.

Además de ofrecer métodos para inicializar y modificar todas las variables de control, se encarga de verificar las grabaciones correctas de datos en cada uno de los distintos ficheros y de mantener coherencia en la aplicación.

Básicamente se dedica a gestionar las peticiones del usuario, representadas como mensajes desde el interfaz gráfico de usuario, solicitando u ofreciendo a quien corresponda métodos o datos conforme al diagrama de clases. Sin embargo sí hay métodos que merecen ser comentados más profundamente:

Generar una lista de tareas hardware

La generación de tareas se basa en la clase Random de C++. Partiendo de los parámetros introducidos por el usuario, se genera una lista de tareas de la longitud indicada y acotada por los límites seleccionados: para cada parámetro se produce un número aleatorio mientras no esté comprendido entre dichos límites.

Al generar constantemente números aleatorios que deben estar comprendidos en cierto rango, se corre el riesgo de que el cómputo no termine, esto es, que se quede "colgado". Se ha optimizado dicho proceso generando un número comprendido entre cero y el máximo (método predefinido random(int)), si este valor no es mayor que el límite inferior, se le suma dicho límite y se comprueba si ahora está contenido en el intervalo deseado. En cualquier caso, siempre que la diferencia entre límites sea medianamente significativa el proceso terminará con éxito y en un tiempo prácticamente inapreciable.

En el momento en que todas las tareas han sido inicializadas, la lista pasa a estar disponible para su reubicación por el algoritmo ya que las posiciones que deben ocupar se fijan a -1.

Entrada/salida desde fichero.

La clase encargada de cargar datos desde fichero o bien guardarlos es Fichero, sin embargo, los métodos que ofrece solamente devuelven una lista de cadenas leída desde el archivo indicado o bien escribe la que recibe, siendo la clase Interfaz la encargada de procesar los datos, en asociación con UnitPrincipal(GUI) que es quien refleja las elecciones del usuario.

En primer lugar, para abrir un fichero utilizamos el componente ofrecido por Borland TOpenDialog, que permite al usuario seleccionar uno entre los distintos ficheros almacenados en disco y desplazarse cómodamente por los distintos directorios. Para facilitar distinguir los ficheros de cada uno de los tipos, hemos forzado la propiedad FILTER de estos cuadros de diálogo de modo que por defecto solamente muestren los archivos cuya extensión coincide con la predefinida en cada caso de uso del programa, si bien, con el fin de aumentar la flexibilidad, también se permite el típico "Todos los archivos".

El directorio por defecto será aquel donde esté corriendo la aplicación.

El componente TSaveDialog será el que se emplee en las operaciones complementarias de guardar en fichero: similar al TOpenDialog sólo restaría comentar las precauciones tomadas con los nombres de fichero escogidos por el usuario:

Pese a los distintos tipos predefinidos, se permite que el usuario escoja el nombre de sus ficheros, siempre que tengan alguna extensión, si esto no fuese así, por defecto se añaden los tres caracteres representativos del archivo.

Antes de comentar los distintos tipos de archivo, es justo dedicar unas líneas al control TStringList, empleado como archivador de todos los textos que se manejan en la aplicación.

Se ha elegido este control, derivado de Tstring, por las facilidades que ofrece, es una tabla de cadenas donde guardamos en cada fila una cadena de caracteres que representa un determinado elemento de interés (FPGA o tarea HW). Como quiera que a cada una de estas líneas se puede acceder mediante un índice, es prácticamente inmediato la elección de la componente requerida como si de un array se tratara.

Además, los métodos LoadFromFile (nombreFichero) y SaveToFile (nombreFichero) permiten un rápido y fácil acceso a disco en las operaciones de lectura y escritura.

Veamos ahora los tres tipos de archivo individualmente

Archivos históricos.

Para conseguir un archivo histórico, principal funcionalidad de la aplicación, deberemos tener cargadas tanto una FPGA como una lista de tareas hardware, así como un algoritmo (ver manual de usuario), entonces ordenaremos que el algoritmo coloque dichas tareas en función de los criterios elegidos. Una vez tenemos la lista de tareas con los datos de cada una de ellas ya procesada podemos pasar a la simulación, es decir, se ha conseguido un histórico.

La extensión por defecto de un fichero histórico es HST, si bien también se oferta la posibilidad de guardarlo con cualquier otra que el usuario prefiera. Un archivo .hst se compone de tres partes claramente diferenciadas, aunque no es recomendable su edición:

- Cabecera:
  Información necesaria para que la aplicación pueda generar correctamente una reproducción. Consta de cuatro líneas:

  - Primera línea: Comentario legible para cualquier usuario que indica el algoritmo y la FPGA sobre la que se aplicó
  - Segunda línea: modo de trabajo (Online/Offline)
  - Tercera línea: identificador del algoritmo para la aplicación.
  - Cuarta línea: FPGA (nombreFPGA anchoFPGA largoFPGA)

- Tareas procesadas:
  Lista de tareas preprocesadas, también consta de una cabecera explicativa sobre los datos guardados y luego dichos datos:
  Ancho, largo, tiempo de llegada, instante máximo permitido para que finalice la tarea, tiempo de ejecución, colores (cantidad de rojo, verde y azul que se le asigno al color de la tarea) y posiciones en las que se colocará: X y Z forman el plano e Y la altura.
  Cada tarea ocupa una línea.

- Comentarios:

  Los mensajes generados por el algoritmo se grabarán al final del archivo como texto llano, sin comprobar su formato.
  Un * separa la lista de tareas de los comentarios generados por el algoritmo, que cierran el fichero.

Un ejemplo de archivo histórico podría ser el siguiente:

```
Histórico resultante de aplicar el algoritmo Prueba sobre la FPGA peterMorbius en modo Offline
Off
1
peterMorbius 12 20
```

```
        Ancho Largo TiempoLlegada TiempoMaximo TiempoEjecucion Rojo Verde Azul PosicionX PosicionYB PosicionYT PosicionZ
        3 5 2 30 14 0,565656542778015 0,121212124824524 0,828282833099365 0 0 14 0
        5 5 2 30 12 0,585858583450317 0,161616161465645 0,252525240182877 0 0 12 3
        4 5 15 30 11 0,292929291725159 0,979797959327698 0,929292917251587 0 0 11 8
        4 5 4 30 4 0,575757563114166 0,535353541374207 0,909090936183929 5 0 4 0
        5 5 5 30 17 0,353535354137421 0,505050480365753 0,787878811359406 5 0 17 4
        5 4 5 30 15 0,242424249649048 0,535353541374207 0,838383853435516 10 0 15 0
        4 3 10 30 14 0,585858583450317 0,121212124824524 0,494949489831924 10 0 14 5
        4 5 4 30 7 0,535353541374207 0,838383853435516 0,565656542778015 13 0 7 5
        5 4 12 30 7 0,686868667602539 0,151515156030655 0,272727280855179 14 0 7 0
        3 3 14 30 5 0,161616161465645 0,69696968793869 0,848484873771667 5 0 5 9
        3 4 8 30 20 0,65656566619873 0,171717166900635 0,838383853435516 8 0 20 9
        4 3 10 30 7 0,616161644458771 0,767676770687103 0,171717166900635 5 4 11 0
        3 4 14 30 9 0,232323229312897 0,585858583450317 0,898989915847778 12 0 9 9
        4 4 1 30 25 0,888888895511627 0,585858583450317 0,404040396213531 13 7 32 5
        4 3 7 30 22 0,363636374473572 0,828282833099365 0,949494957923889 14 7 29 0
        *
        Tarea insertada en la posicion (0,0,0)
        Tarea insertada en la posicion (3,0,0)
        Tarea insertada en la posicion (8,0,0)
        Tarea insertada en la posicion (0,5,0)
        Tarea insertada en la posicion (4,5,0)
        Tarea insertada en la posicion (0,10,0)
        Tarea insertada en la posicion (5,10,0)
        Tarea insertada en la posicion (5,13,0)
        Tarea insertada en la posicion (0,14,0)
        Tarea insertada en la posicion (9,5,0)
        Tarea insertada en la posicion (9,8,0)
        Tarea insertada en la posicion (0,5,4)
        Tarea insertada en la posicion (9,12,0)
        Tarea insertada en la posicion (5,13,7)
        Tarea insertada en la posicion (0,14,7)
```

Para cargar un histórico, el archivo elegido debe tener el formato comentado anteriormente; se procesará la cabecera ignorando la primera línea (datos para reconocimiento) y saltando directamente a fijar el modo, el algoritmo, la FPGA y la lista de tareas ya procesada.

Archivos de FPGA

Una FPGA está representada como un nombre y dos dimensiones, opcionalmente se le puede asociar un comentario. Para crearla, únicamente se deben rellenar los campos asociados en la ventana correspondiente atendiendo a las restricciones: el nombre no debe contener espacios en blanco y las dimensiones han de estar comprendidas en el intervalo cerrado [1,100]. El comentario asociado nunca ha de comenzar con una línea en blanco.

Al crear una nueva, se ofrece la posibilidad de guardarla, si esto fuese así, se volcaría a un fichero empleando el apoyo del ya mencionado SaveDialog; sin embargo, el formato elegido para los archivos de FPGAs es el TXT con el fin de una fácil edición, ya sea en un cambio de dimensiones o una modificación del comentario correspondiente o incluso crear directamente desde un editor de texto varias. El "volcado" sobre fichero .TXT consistiría en añadir tanto la FPGA como el comentario al final del mismo. Queda como responsabilidad del usuario permitir repeticiones de FPGAs. Si el fichero elegido para contener la nueva FPGA no existiese, se crearía uno nuevo con ella.

Un ejemplo de fichero de FPGA:

```
Esto es un archivo de FPGAs
XC2S15 12 26
*FPGA de la familia Spartan II con15000 compuertas equivalentes (System Gates)
XCV3200C 20 42
*FPGA de la familia VIRTEX II con 4047000 compuertas equivalentes (System Gates) de
*2.5 voltios
*Empleada en aplicaciones de reconocimiento del habla
VirtexE2000 34 78
Este comentario no será procesado....
inventada 12 34
*Puedo escribir los versos más tristes esta noche
*Escribir, por ejemplo, la noche está estrellada y tiritan, azules, los astros, a lo lejos
*el viento de la noche gira en el cielo y canta
*puedo escribir los versos mas tristes esta noche
```

Como ya se ha comentado, los archivos que albergan las FPGAs son sencillos ficheros de texto, de cara a favorecer su posterior (o incluso anterior) edición, lo cual conlleva cierto riesgo a la hora de procesarlo para cargar las tareas. Se abre también la posibilidad de que el usuario introduzca comentarios que no estén asociados a ninguna

de las FPGAs almacenadas, estos comentarios no tienen ningún indicativo y serían simples líneas de texto intercaladas entre las distintas FPGAs.

El archivo se interpreta de la siguiente manera: se leería línea a línea, comprobando que la cadena leída cada vez es una FPGA válida, en caso de encontrar una se pasa a asociar todas las líneas siguientes como comentario siempre que empiecen como * de tal forma que en cuanto se detecta un comienzo de línea distinto se da la FPGA por cargada.

Al elegir un fichero de FPGAs para escoger una, se vuelca el contenido a la memoria en dos objetos de la clase TStringList, uno contendrá las FPGAs y el otro los comentarios asociados a cada una de ellas; de este modo conseguimos un rápido acceso a cada una de las FPGAs válidas cargadas y sus comentarios gracias a los métodos ofrecidos por TStringList. Se ha optado por dejar residente el contenido del fichero en la memoria del programa hasta que el usuario decida cargar otra vez un archivo distinto, dando siempre la opción de cambiar de FPGA sin tener que acceder a disco e incrementando claramente el rendimiento al reducir el número de accesos a memoria secundaria.

Archivos de tareas

La lista generada también se guarda mediante SaveDialog en fichero. De manera similar a los históricos, un archivo de tareas consiste en una cabecera con texto y la lista de tareas sin procesar, o lo que es lo mismo, para cada tarea (o en cada línea) se escribe su ancho, largo, tiempo que tarda en ejecutarse, tiempo en el que llega e instante máximo para que finalice.

La extensión asociada a estos ficheros es TSK y la manera de interpretarlos es similar a los de FPGA por si el usuario quiere introducir algún comentario: cada línea a partir de la cabecera se intenta interpretar como una tarea, si lo es se añade a la lista y si no lo es se ignora.

Un ejemplo sería:

```
Archivo de tareas...
CODIFICACIÓN: Ancho Largo Ejecución Llegada Máximo
 A--L--E-Ll-M
11 9 8 7 30
12 4 10 12 30
7 2 5 21 30
6 12 15 11 30
4 11 22 6 30
10 2 7 2 30
8 10 19 1 30
1 1 16 7 30
11 1 12 11 30
6 1 20 2 30
```

Representación gráfica de los resultados y del código o mensajes generados por el algoritmo.

El texto que devuelve el algoritmo se muestra en una ventana colocada a la izquierda de la simulación. Dicha ventana es un componente de tipo Tmemo ofertado por Borland q recibe una lista de cadenas contenida en un objeto de la ya comentada clase TStringList. Se ha escogido esta clase por las facilidades para añadir cadenas de caracteres así como su posterior tratamiento.

Quizá un método que merezca ser comentado aparte es Interfaz::borraFPGA(int) que elimina del fichero cargado en memoria la FPGA asociada al índice que recibe como parámetro. Dicho método elimina tanto la FPGA como su comentario asociado de sendos TStringList y luego procede a machacar el fichero original, lo que conlleva que los comentarios creados desde un editor de textos sean borrados, si se quieren mantener estos comentarios es recomendable editar el archivo de texto y borrar "a mano". Esta decisión de diseño viene motivada por dar mayor flexibilidad al archivo de FPGAs, puesto que al poder analizar cualquier .TXT y permitir repeticiones en sus definiciones, el proceso de eliminar la FPGA y respetar absolutamente todo lo demás sería altamente costoso.

La "comunicación" del algoritmo en modo texto se basa en el método Add(String) de la clase TstringList que añade la línea que recibe como parámetro al final de la tabla, con lo que es sencillo comentar paso a paso la ejecución (véase Algoritmo). Y se vuelca directamente sobre el Memo de históricos que aparece en la GUI.

Visualización 3D

En la parte de visualización nos ocupamos de la generación de imágenes de una FPGA y sus tareas a través de librerías gráficas, en nuestro caso hemos elegido usar la librería gráfica Open GL (Open Graphics Library).

Open GL es una herramienta software en forma de librería C que permite la comunicación entre el programador y el hardware de la máquina para el diseño de gráficos. Dentro de las características que tiene Open GL podemos mencionar que es portable, no ofrece comandos para la manipulación de ventanas, ni para leer datos introducidos por un usuario; tampoco dispone de comandos de alto nivel para describir escenas 3D, en Open GL es necesario construir los gráficos a partir de sus primitivas geométricas: puntos, líneas y polígonos. Por lo cual junto con Open GL se distribuye siempre la librería GLU (Open GL Utility Library), que esta construida a partir de Open GL y suministra comandos de alto nivel para el dibujo de gráficos 3D, así mismo, para la manipulación de ventanas y E/S puede utilizarse la librería GLUT (Open GL Utility ToolKit), que tiene la ventaja de ser portable, pero en contrapartida tiene que el desarrollo de interfaces gráficas de usuario es complejo.

Open GL es una máquina de estados en la que tenemos una colección de variables de estado a las que vamos cambiando su valor.

Nosotros utilizaremos el entorno de desarrollo C++ Builder puesto que nos permite desarrollar de interfaces gráficos de usuario de manera menos compleja.

A continuación explicaremos la estructura que ha de tener un programa en C++ con Open GL, nuestra clase o unidad principal consta de una serie de métodos que son necesarios para la manipulación del hardware, en este caso la tarjeta gráfica, y de esta manera sea posible visualizar la simulación 3D de una FPGA.

1. En la UnitPrincipal.h , que es la declaración de la clase principal, incluimos las directivas :

```
#include <gl\gl.h>
#include <gl\glu.h>
```

que cargan las librerías de Open GL y GLU , que se encuentran en c:\Archivos de Programa\Borland\Cbuilder5\Include.

La implementacion de los .h son dll's de Windows, en concreto en c:\WinNT\system32 se encuentran OPENGL32.dll y glu32.dll

Dentro de la clase tendremos métodos públicos asociados a eventos de formularios que se pueden gestionar a partir del entorno gráfico.

```
class TGLForm3D : public TForm
{
__published:
    .
    .
    .
  void __fastcall FormResize(TObject *Sender);
  void __fastcall FormPaint(TObject *Sender);
  void __fastcall FormDestroy(TObject *Sender);
  void __fastcall FormCreate(TObject *Sender);
```

- FormResize: Es un método asociado al evento OnResize del formulario. Se define siempre que deseemos cambiar el tamaño de la ventana, para lo cual es necesario definir el sistema de coordenadas (coordenadas mundiales) a utilizar en la ventana (ventana del mundo) y el puerto de vista , que es la parte de la ventana del mundo en la que saldrá dibujada la imagen. En nuestro caso la imagen sale dibujada hacia la parte derecha de la pantalla.

- FormPaint: Este método va asociado al evento OnPaint del formulario. Necesariamente se define siempre y debe contener las instrucciones necesarias para dibujar el gráfico.

- FormDestroy: Método asociado al evento OnDestroy del formulario, destruye todos los objetos credos liberando memoria.

- FormCreate: Método asociado al evento OnCreate. Se ejecuta cuando se crea el formulario y es aquí donde se inicializan las variables de la clase.

Seguidamente tendremos la declaración de variables privadas donde distinguimos las siguientes variables:

```
private:        // User declarations
   HDC hdc;
   HGLRC hrc;
   GLfloat w, h, nRange;
```

- HDC: (Handle device Context) Fija el área cliente de la ventana en la que pintamos.
- HGLRC: (Handle Open GL Rendering context) Fija el contexto en el que Open GL dibuja sus gráficos.
- w,h : son la anchura y altura del formulario

Dentro de los métodos privados a destacar estan:

```
   void __fastcall SetPixelFormatDescriptor();
   void __fastcall GLScene();
```

- SetPixelFormatDescriptor: Se encarga de seleccionar las características del píxel ajustándose a las posibilidades del sistema.
- GLScene: Es el método que contiene las instrucciones Open GL necesarias para dibujar la FPGA y sus tareas.

Por último comentamos lo que es extern dentro de UnitPrincipal.h

```
extern PACKAGE TGLForm3D *GLForm3D;
```

Extern: nos indica que la variable GLForm3D, que es el nombre del formulario, se definirá fuera, en UnitPrincipal.h sólo indicamos que dicha variable existe, pero no reservamos memoria. Para usarla hay que definirla en el .cpp del formulario.

2. Ahora pasaremos a comentar los métodos implementados en UnitPrincipal.cpp relacionados con Open GL:

```
void __fastcall TGLForm3D::FormCreate(TObject *Sender)
{
  hdc = GetDC(Handle);
  SetPixelFormatDescriptor();
  hrc = wglCreateContext(hdc);
  if(hrc == NULL)
    ShowMessage("Error CreateContex");
```

```
  if(wglMakeCurrent(hdc, hrc) == false)
    ShowMessage("Error MakeCurrent");

  w = ClientWidth;
  h = ClientHeight;
  glEnable(GL_LIGHTING);
  glEnable(GL_LIGHT0);
  GLfloat LuzDifusa[]={1.0,1.0,1.0,1.0};
  glLightfv(GL_LIGHT0, GL_DIFFUSE,LuzDifusa);
  GLfloat LuzAmbiente[]={0.5,0.5,0.5,1.0};
  glLightfv(GL_LIGHT0, GL_AMBIENT, LuzAmbiente);
  glEnable(GL_COLOR_MATERIAL);
  glShadeModel(GL_SMOOTH);   //defecto
  glEnable(GL_DEPTH_TEST);
  glEnable(GL_NORMALIZE);

  // inicializaciones de variables
}
```

- hdc = GetDC(Handle): es una funcion del API de Windows, toma como parémetro Handle, que es el identificador interno del formulario, es decir, una propiedad solo de lectura del formulario.
  Este método devuelve el andel (andel ddevice context) del área cliente (área de dibujo) de la ventana especificada por Handle.
  Una vez fijado hdc, cualquier llamada a una función GDI (Graphics Device Interface) dibujará en el área cliente referenciada por hdc.
  GDI, interfaz de dispositivos gráficos suministrada por Windows. Suministra funciones para dibujar gráficos de forma independiente del dispositivo. A estas funciones del API de Windows se las puede llamar directamente desde C++. Pero C++ encapsula estas funciones gráficas de forma fácil y eficiente.

- SetPixelFormatDescriptor(): Fija el formato de pixel a utilizar en el área cliente de la ventana especificada por hdc.

- hrc = wglCreateContext(hdc): Toma como parámetro el área del cliente de la ventana especificado por hdc y creo un Open GL rendering context, necesario para la ejecución de comandos Open GL.
  Open GL rendering context es un puerto a través del cual pasan todos los comandos Open GL. Hay que puntualizar que no es lo mismo un 'rendering context' que un 'device context', el primero contiene información relativa a Open GL y el segundo contiene información relativa a las GDI's.

- hrc = wglCreateContext(hdc): hrc tiene el mismo formato de pixel que hdc. El método wglCreateContext puede tener exito o fallar, Si falla, es decir que no puede crearse un contexto para Open GL entonces devuelve NULL.

- (wglMakeCurrent(hdc, hrc): pone a hrc como el puerto actual a través del cual pasan todos los comandos Open GL (rendering context).

- Cuando hrc == NULL, se desactiva el puerto actual y se libera el hdc asociado.

- ClientWidth y ClientHeight son propiedades de tipo published de lectura y escritura del formulario, se gestionan partir del entorno gráfico.

- glEnable(GL_LIGHTING): Activa el modelo de iluminación.

- glEnable(GL_LIGHT0): Define una fuente de luz

- GLfloat LuzDifusa[]={1.0,1.0,1.0,1.0};
  glLightfv(GL_LIGHT0, GL_DIFFUSE,LuzDifusa);
  GLfloat LuzAmbiente[]={0.5,0.5,0.5,1.0};
  glLightfv(GL_LIGHT0, GL_AMBIENT, LuzAmbiente);

  con estos 4 comandos definimos la intensidad de luz difusa y de ambiente que habrá en el escenario, se especifica mediante un color RGBA {r,g,b,a}

- glEnable(GL_COLOR_MATERIAL): Activa la definición de colores de materiales usados en los gráficos, los colores especificados por el comando glColor(r,g,b,a) serán tomados como propiedades del material.

```
void __fastcall TGLForm3D::SetPixelFormatDescriptor()
{
  PIXELFORMATDESCRIPTOR pfd = {
        sizeof(PIXELFORMATDESCRIPTOR),
      1,
      PFD_DRAW_TO_WINDOW | PFD_SUPPORT_OPENGL |
PFD_DOUBLEBUFFER,
      PFD_TYPE_RGBA,
      24,
      0,0,0,0,0,0,
      0,0,
      0,0,0,0,0,
      32,
      0,
      0,
      PFD_MAIN_PLANE,
      0,
      0,0,0 };
  int PixelFormat = ChoosePixelFormat(hdc, &pfd);
  SetPixelFormat(hdc, PixelFormat, &pfd);
}
```

- PIXELFORMATDESCRIPTOR: Es una estructura que describe el formato de pixel de una superficie de dibujo. Sus campos son los siguientes:

  - El primer campo especifica el tamaño de la estructura por lo que será siempre sizeof(PIXELFORMATDESCRIPTOR)

  - El segundo campo es la versión de la estructura por lo que será siempre 1

  - El tercer campo determinan las propiedades del buffer de pixeles:

    - PFD_DRAW_TO_WINDOW: El buffer debe dibujarse en una ventana.
    - PFD_SUPPORT_OPENGL: El buffer debe soportar comandos Open GL.
    - PFD_DOUBLEBUFFER: Tenemos dos buffers.

  - PFD_TYPE_RGBA: El color de cada píxel se determina siguiendo el modelo RGBA o RGB más Alfa, donde Alfa indica el grado de transparencia.

  - 24 y 32: especifican respectivamente, la profuncida de color (8 bits/ pixel = color verdadero) y el buffer de profundidad para el eje de las z's.

  - Los campos con valor 0 son opciones no utilizadas por el ChoosePixelFormat y PFD_MAIN_PLANE, es un campo obligatorio.

- int PixelFormat = ChoosePixelFormat(hdc, &pfd): La función ChoosePixelFormat se encarga de encontrar un formato de píxel soportado por el hdc, se además sea el mejor que se ajuste a las especificación dada por el pfd.

- SetPixelFormat: Establece en hdc el formato de pixel dado por PixelFormat.

```
void __fastcall TGLForm3D::FormResize(TObject *Sender)
{
  w = ClientWidth;
  h = ClientHeight;
  if (h==0.0) h=1.0;
  if (w==0.0) w=1.0;

  glViewport(100,0, w, h);
  if(perspectiva)
    camara.setPerspectiva(30.0,w/h,0.1,400.0);
  else{
      if (w <= h)
        {//nRange = w/8;
          camara.setOrtogonal(-nRange,nRange,-nRange*h/w,nRange*h/w,-
nRange*2,
```

```
                nRange*2);
        }
       else
        {//nRange = h/8;
        camara.setOrtogonal(-nRange*w/h,nRange*w/h,-nRange,nRange,-
nRange*2,
                nRange*2);
        }
  };
   if(perspectiva){
    ojo.modPunto(130,50,130);
    look.modPunto(0,0,0);
  }
  else{
    ojo.modPunto(10,10,10);
    look.modPunto(0,0,0);
  };
  camara.setMarco(ojo,look,Vector3D(0,1,0));

  GLScene();
}
```

- En el método FormResize establecemos el puerto de vista para nuestra simulación y utilizamos la cámara para tener la opción de poder visualizar nuestras imágenes en dos proyecciones diferentes: ortogonal y perspectiva. De estos tipos de proyecciones hablaremos más adelante con detalle.

- El puerto de vista lo determina el comando glViewport(100,0,w,h), es decir, establecemos la ventana en la que vamos a dibujar la FPGA y las tareas. (100,0) indica la esquina inferior izquierda del rectángulo y (w,h) indica la posición de la esquina superior derecha de la ventana. Ambas vienen dadas en coordenadas de la ventana.

```
void __fastcall TGLForm3D::GLScene()
{
  glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
  GLfloat LuzPosicion[]={25.0,25.0,0.0,1.0};
  glLightfv(GL_LIGHT0, GL_POSITION,LuzPosicion);

  glEnable(GL_LINE_SMOOTH);
  glEnable(GL_BLEND);
  glBlendFunc(GL_SRC_ALPHA,GL_ONE_MINUS_SRC_ALPHA);

  if(visualizar)
    drawEscena();

  glDisable(GL_BLEND);
```

```
  glDisable(GL_LINE_SMOOTH);

  SwapBuffers(hdc);
}
```

- Este método GLScene contiene las instrucciones Open GL necesarias para el dibujo de nuestra simulación

- glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT): Esta primera instrucción vacía el buffer donde se encuentra almacenada la imagen.

- Los comandos GLfloat LuzPosicion[]={25.0,25.0,0.0,1.0} y glLightfv(GL_LIGHT0, GL_POSITION,LuzPosicion) determinan la posición y la dirección de la fuente de luz que ilumina nuestros gráficos.

- Los comandos que mencionamos a continuación realizan un suavizado sobre las lineas que dibujamos.

```
glEnable(GL_BLEND)
glBlendFunc(GL_SRC_ALPHA,GL_ONE_MINUS_SRC_ALPHA)
```

y luego es deshabilitado con el comando:

```
  glDisable(GL_BLEND);
```

- SwapBuffers(hdc): Lleva a cabo el intercambio de buffers, tenemos un sistema de doble buffer.

- drawEscena(): Método que dibuja toda la simulación.

```
void __fastcall TGLForm3D::FormDestroy(TObject *Sender)
{
  visualizar=false;
  ReleaseDC(Handle,hdc);
  wglMakeCurrent(NULL, NULL);
  wglDeleteContext(hrc);
}
```

- Con este método liberamos recursos utilizados.

3. Para obtener una simulación más real, hemos implementado la clase Camara, ya que Open GL trabaja internamente con matrices, concretamente con la matriz de proyección, la de modelado y vista, y la de puerto de vista.
La clase contiene métodos que hacen posible tener en la simulación de la FPGA con la posibilidad de elegir distintas vistas, así como dos proyecciones diferentes, rotación de la escena que se muestre en casa momento y también ofrece la posibilidad de hacer un zoom de acercamiento o alejamiento de la imagen.

No olvidemos mencionar que en la clase Camara trabajamos con las matrices antes mencionadas:

- Matriz de Proyección: almacena la forma en la que debe proyectarse el gráfico en el volumen de vista.
- Matriz de modelado y vista: almacena las transformaciones de rotación, traslación y escalación.
- Matriz de puerto de vista: almacena cómo dibujar el gráfico en el puerto de vista.

La cámara (u ojo) tiene su propio sistema de coordenadas, este sistema define lo que llamaremos marco de la cámara y determina donde estará situado el ojo de la cámara, a efectos prácticos, la posición del ojo de la cámara da lugar a las vistas, zoom...etc. A continuación veremos como están implementados los métodos de esta clase:

```
void Camara::setMarco(Punto3D ojo,Punto3D look,Vector3D up){
  float m[16],m2[16];

  glMatrixMode(GL_MODELVIEW);
  glLoadIdentity();
  _ojo=Punto3D(ojo.devX(),ojo.devY(),ojo.devZ());
  _look=Punto3D(look.devX(),look.devY(),look.devZ());
  gluLookAt(_ojo.devX(),_ojo.devY(),_ojo.devZ(),_look.devX(),_look.devY(),
  _look.devZ(),up.devX(),up.devY(),up.devZ());
  glGetFloatv(GL_MODELVIEW_MATRIX,m);
  calculaInversa(m,m2);
  _n.modVector(m2[8],m2[9],m2[10]);
  _v.modVector(m2[4],m2[5],m2[6]);
  _u.modVector(m2[0],m2[1],m2[2]);
};
```

- glMatrixMode(GL_MODELVIEW): carga como matriz actual la matriz de modelado y vista

- glLoadIdentity(): hace la identidad a la anterior matriz.

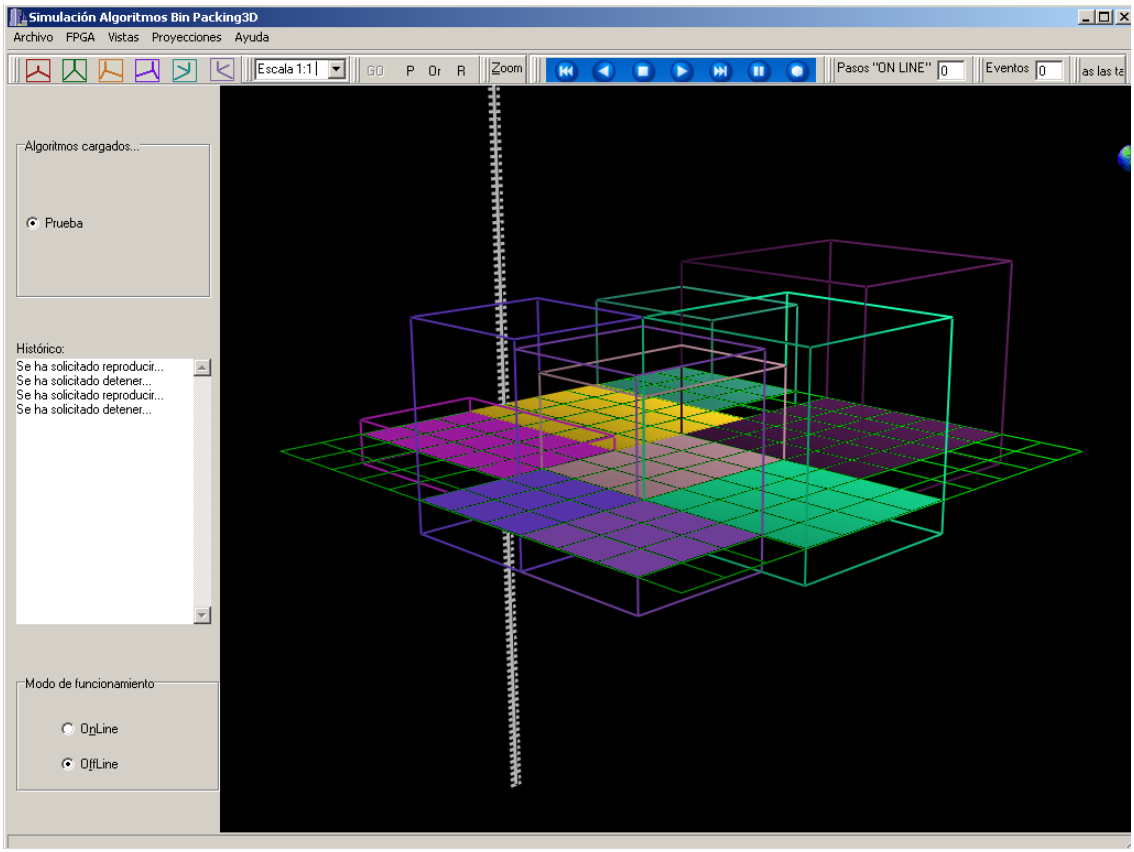
- gluLookAt(ojoX,ojoY,ojoZ,lookX,lookY,lookZ,upX,upY,upZ): define el marco de la cámara (u,v,n,ojo), y la matriz de vista para transformar el sistema

cartesiano al sistema de camara. El marco de la cámara se obtiene a partir de los argumentos de este comando como un sistema ortogonal con origen en 'ojo'. Y la matriz de vista se obtiene como la inversa al marco de la cámara.

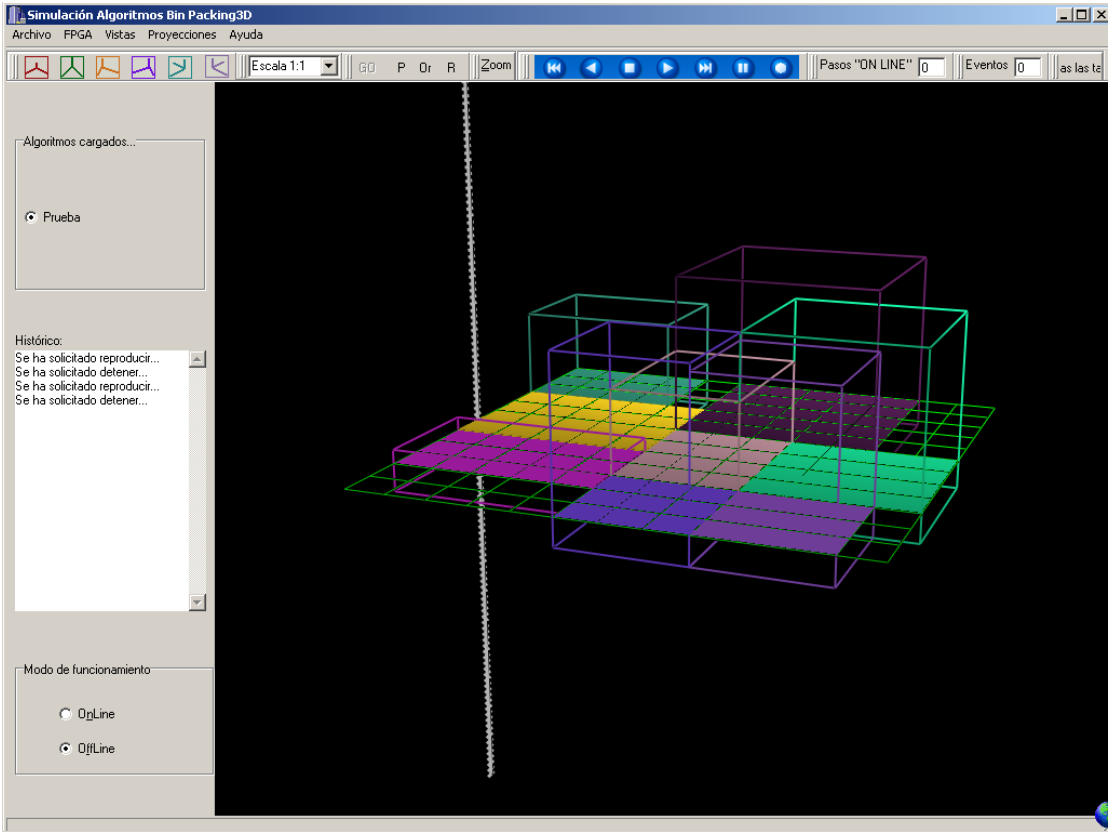
- glGetFloatv(GL_MODELVIEW_MATRIX,m): Consulta la matriz de modelado y vista en curso y la copia en su parámetro m, que es una matriz 4x4.

Al modificar los argumentos ojo, look, up es que podemos tener las distintas vistas de la simulación, así vemos en seguida las distintas vista implementadas todas ellas con una proyección en perspectiva:

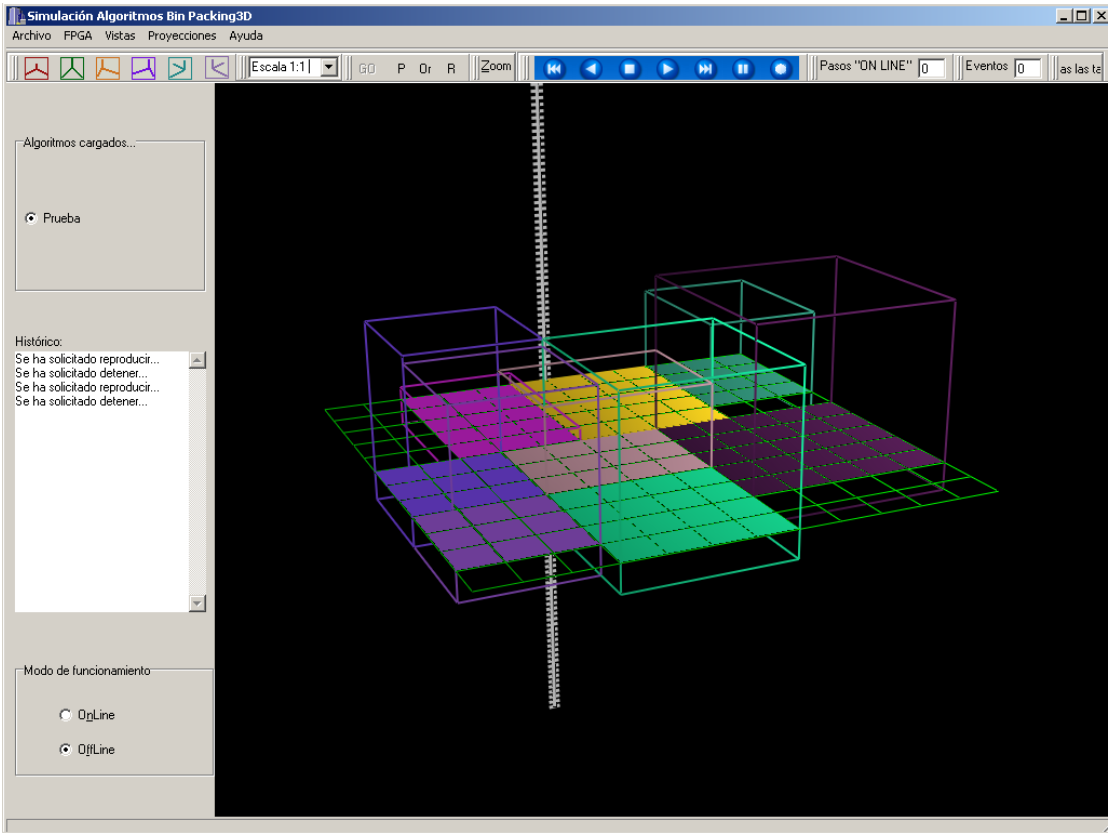
i. VISTA NORMAL EN PERSPECTIVA



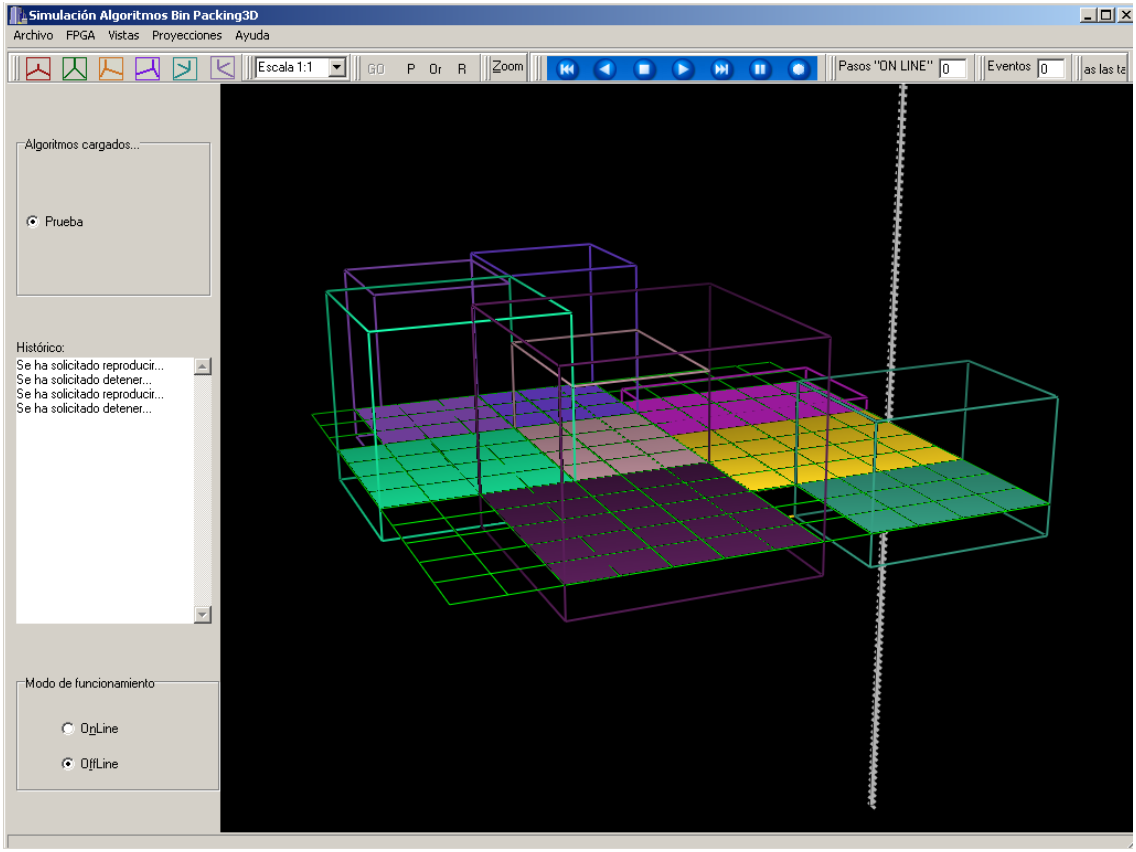
ii. VISTA DESDE EL LATERAL 1 EN PERSPECTIVA



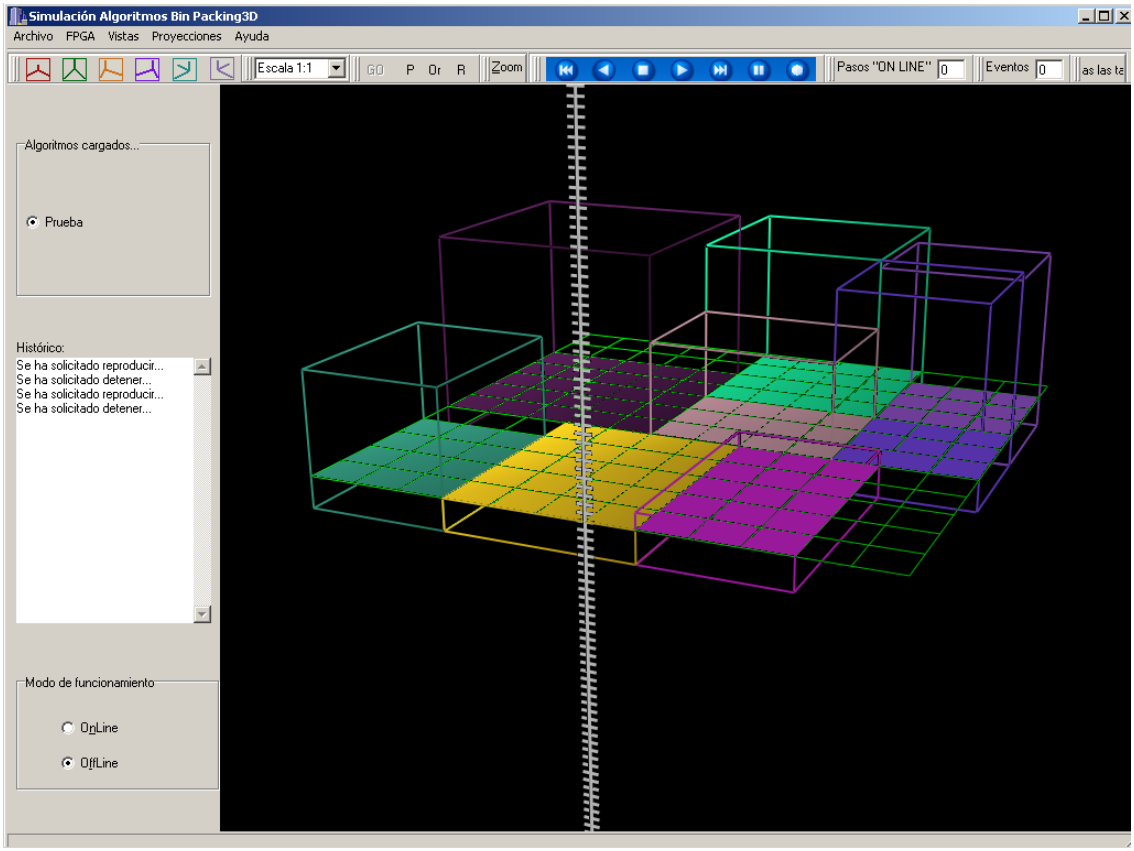
iii. VISTA DESDE EL LATERAL 2 EN PERSPECTIVA



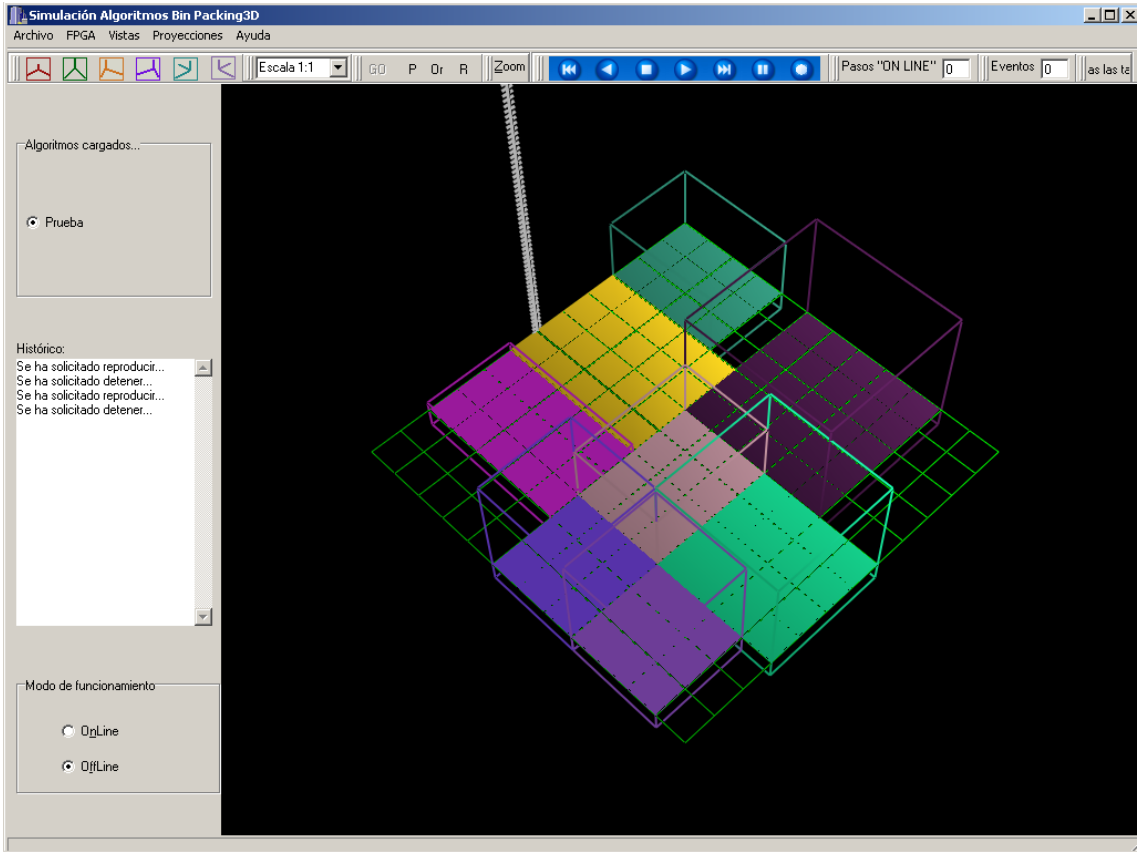
iv. VISTA DESDE EL LATERAL 3 EN PERSPECTIVA



v. VISTA DESDE EL LATERAL 4 EN PERSPECTIVA



vi. VISTA POR ENCIMA EN PERSPECTIVA



Hasta el momento hemos explicado las vistas, ahora veremos como están hechas las proyecciones, habíamos dicho en paginas anteriores que teníamos dos tipos de proyecciones, ortogonal y perspectiva. Las proyecciones son distintas formas de proyectar una imagen en un plano de vista (una ventana) y quedan determinados por la definición de la cámara: marco de coordenadas de la cámara y volumen de vista.

```
void Camara::setOrtogonal(float l,float r,float b,float t,float n,float f){
  glMatrixMode(GL_PROJECTION);
  glLoadIdentity();
  glOrtho(l,r,b,t,n,f);
};
```

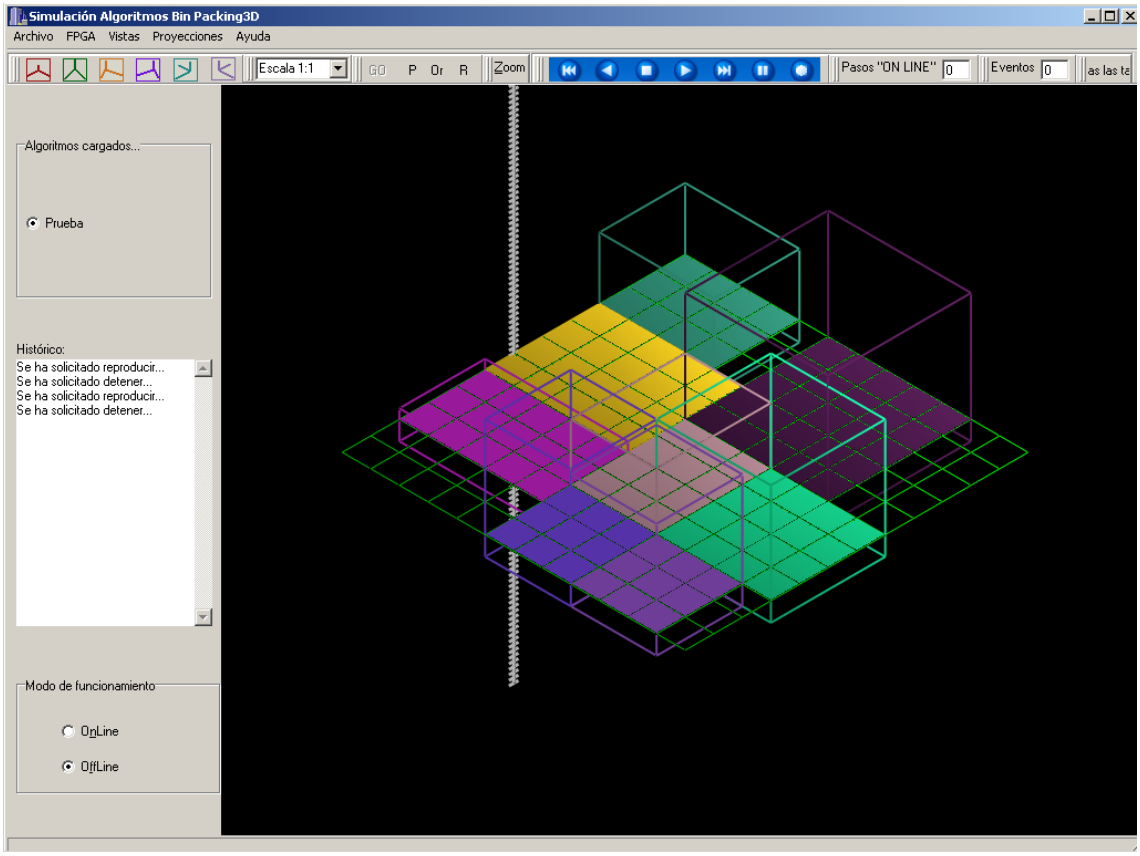
- El método setOrtogonal define la vista ortogonal del gráfico.

- glMatrixMode(GL_PROJECTION): Define la matriz de proyección; junto con los comandos glLoadIdentity() y glOrtho(l,r,b,t,n,f) se determinan, en coordenadas de la cámara, un paralelepípedo alineado con el eje n de la cámara como el volumen de vista de la cámara.

Una muestra de lo que es una vista normal en proyección ortogonal seria la siguiente, en la cual podemos observar que a diferencia de una vista normal en proyección en perspectiva, tanto la parte mas alejada de la imagen como la más cercana

se ven igual; lo cual no sucede en una proyección en perspectiva, donde la parte más alejada de la imagen se ve mas pequeña que la que esta mas cerca.

vii. VISTA NORMAL EN PROYECCIÓN ORTOGONAL

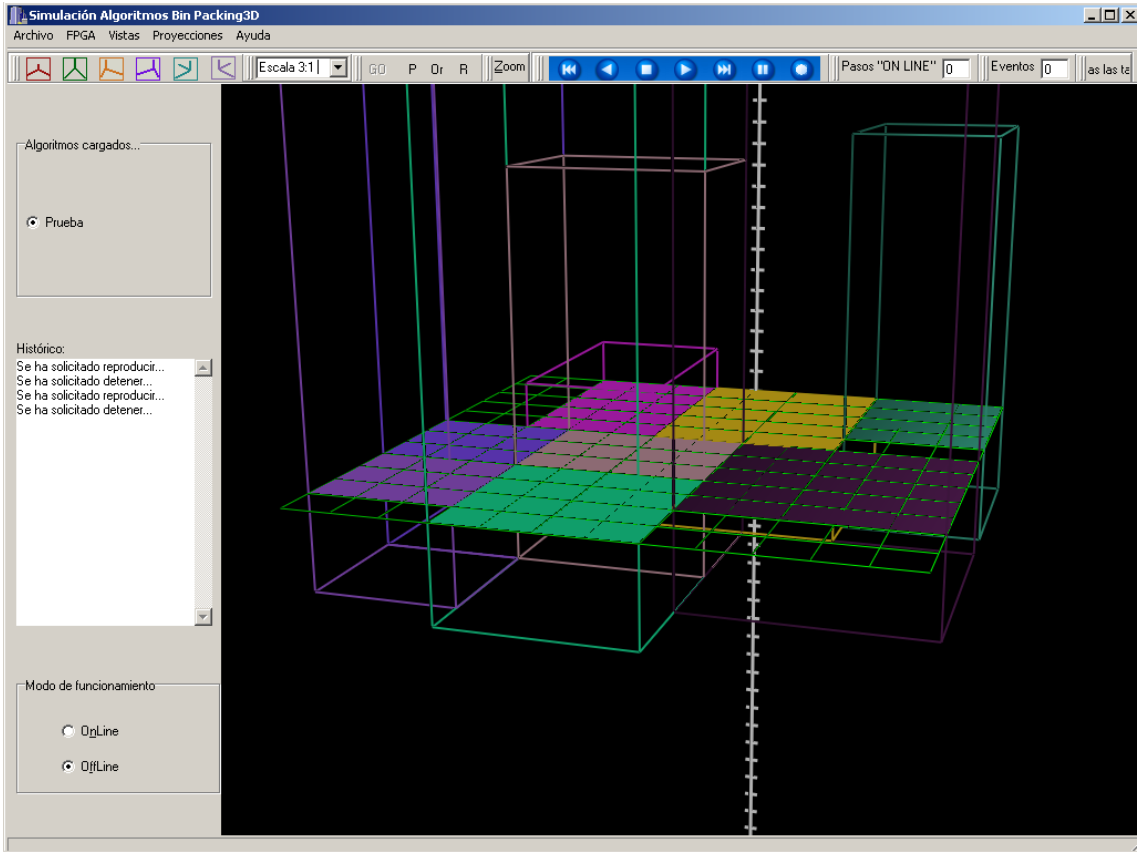



```
void Camara::setPerspectiva(float ang,float ratio,float n,float f){
  glMatrixMode(GL_PROJECTION);
  glLoadIdentity();
  gluPerspective(ang,ratio,n,f);
};
```

- Método que define la perspectiva de un gráfico, definimos el volumen con perspectiva con el comando gluPerspective(ang,ratio,n,f) que define en coordenadas de la cámara (u,v,n,ojo), una pirámide rectangular truncada obtenida a partir de los argumentos del comando.

Sobre las rotaciones de la imagen y cambios de escala en el eje del tiempo, mencionaremos que para ello utilizamos transformaciones afines con los comandos glRotatef(camara.devAngY(),0,1,0) y glScalef(1.0,escala,1.0) respectivamente. Los cuales lo que hacen es dado un punto P en coordenadas del marco actual, multiplica la matriz de modelado y vista por el punto P y nos devuelve el punto P en es sistema de la cámara. Utilizamos también los comandos glPushMatrix() y glPopMatrix() que apilan y desapilan matrices de esta forma evitamos hacer transformaciones inversas una vez hecha la transformación afin.

viii. LA MISMA IMAGEN EN PERSPECTIVA que en i. ESCALA Y CON ROTACIÓN



Para hacer un zoom también ultilizamos métodos de la cámara, en concreto llamamos al método Desplazar():

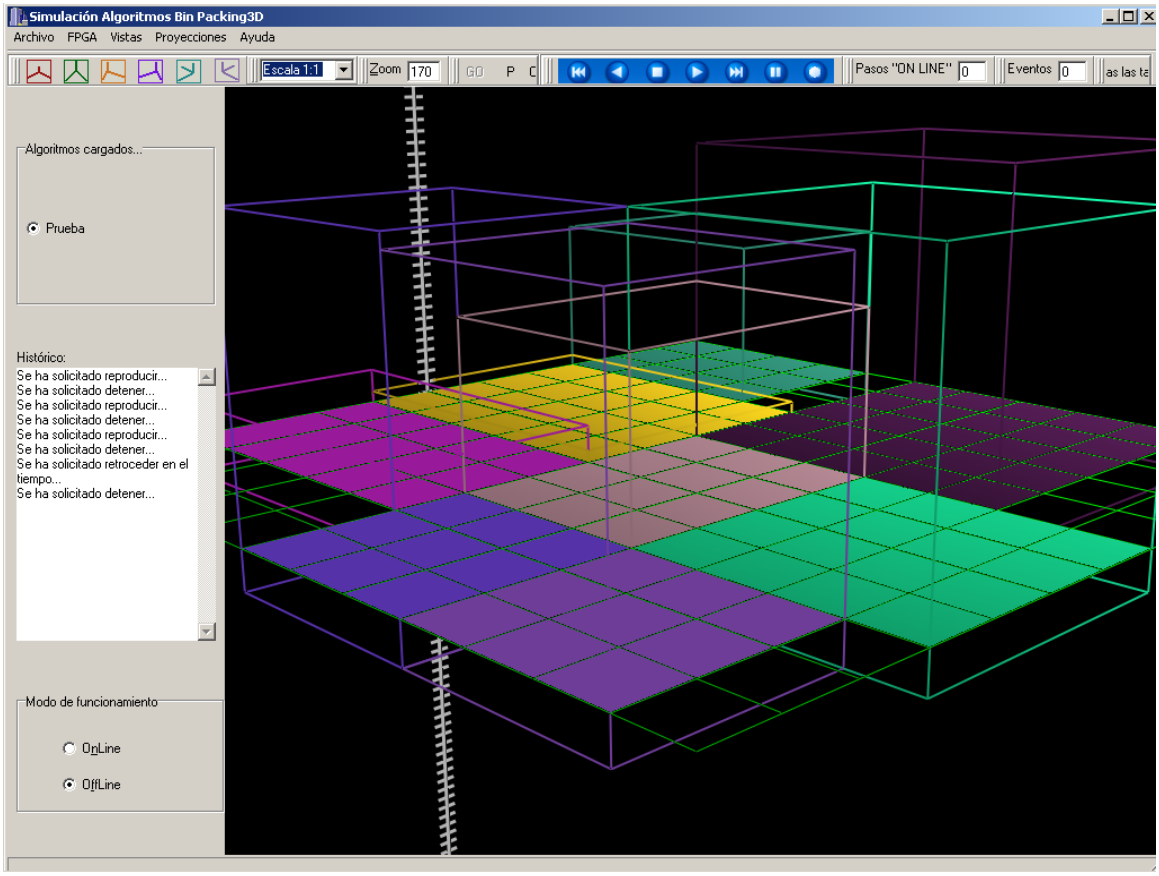
```
void Camara::Desplazar(float distU,float distV,float distN){
  float m[16]={_u.devX(),_u.devY(),_u.devZ(),0,
               _v.devX(),_v.devY(),_v.devZ(),0,
               _n.devX(),_n.devY(),_n.devZ(),0,
               _ojo.devX(),_ojo.devY(),_ojo.devZ(),1};
  float traslacion[16]={1,0,0,0,
                        0,1,0,0,
                        0,0,1,0,
                        distU,distV,distN,1};
  float mv[16],M[16],m2[16];

  glGetFloatv(GL_MODELVIEW_MATRIX,mv);
  mult(m,mv,M);
  mult(m,traslacion,m2);

_ojo.modPunto(_ojo.devX()+distU,_ojo.devY()+distV,_ojo.devZ()+distN);

_look.modPunto(_look.devX()+distU,_look.devY()+distV,_look.devZ()+distN);
```

```
  copia(m2,m);
  calculaInversa(m,m2);
  mult(m2,M,m);
  glMatrixMode(GL_MODELVIEW);
  glLoadMatrixf(m);
};
```

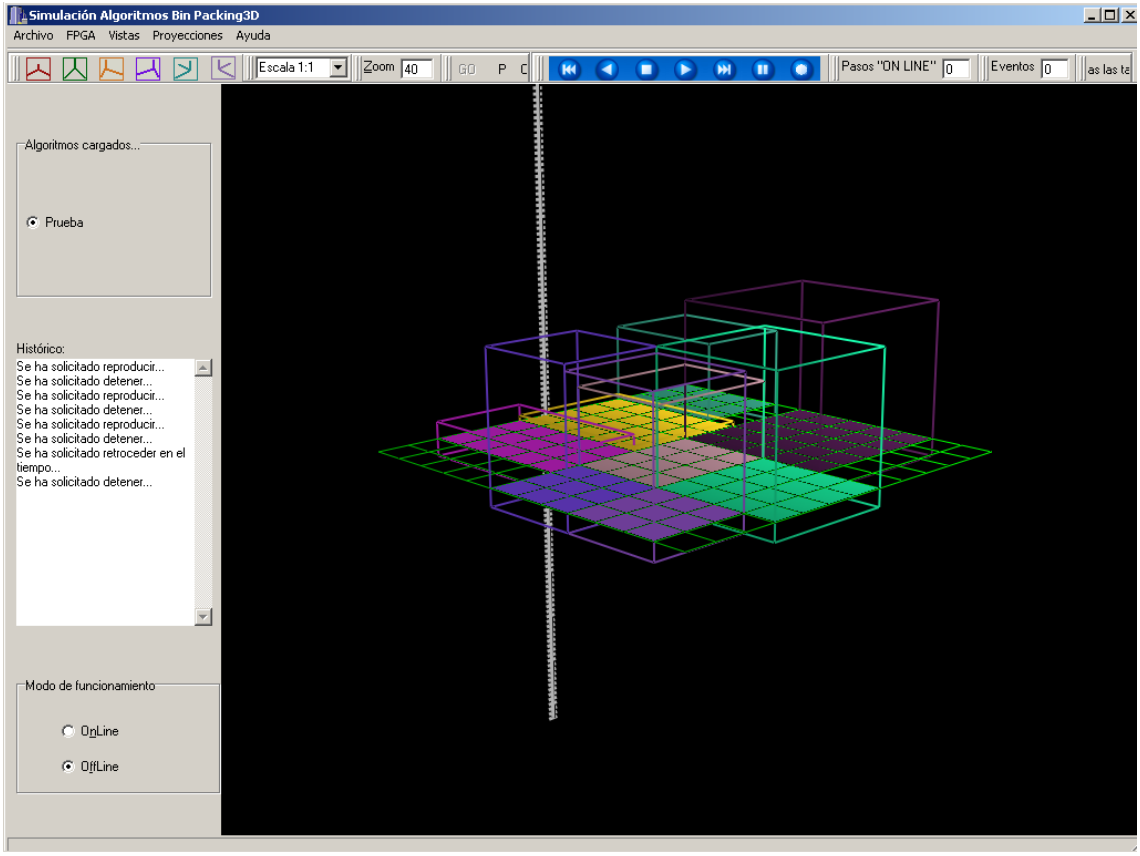
- Los comandos utilizados en este método han sido explicados en páginas anteriores, globalmente podemos decir que lo que hacemos aquí es acercar o alejar el ojo de la cámara una determinada distancia distN respecto al gráfico que tengamos, dando lugar al efecto de zoom. Puesto que el zoom es progresivo, es decir, que si hacemos por ejemplo un zoom al 150%, la imagen se va agrandándose progresivamente hasta llegar al 150%; es necesario hacer llamado a este método un número determinado de veces, estas llamadas las hacemos con la ayuda de un reloj interno (una hebra), que se ejecuta paralelamente al resto de programa.
  Un zoom sólo se puede hacer cuando estamos en perspectiva porque es cuando realmente se ve el efecto de acercamiento o alejamiento.

- Tanto en el zoom , en las rotaciones y en la ejecución de tareas sobre la FPGA tenemos que repintar la imagen para que de el efecto de paso del tiempo, y dado que el repintar actúa sobre buffers internos y sobre la tarjeta grafica del ordendor, estas acciones relentisan la ejecución del programa.

ix. ZOOM AL 170% DE LA IMAGEN ANTERIOR



x. ZOOM AL 40%



Algoritmo

En ésta sección se describirán las clases que intervienen en la ejecución del algoritmo. También se detallará el funcionamiento del algoritmo de ejemplo (cuyo nombre es prueba) que hemos implementado, incluyendo decisiones de diseño, algunas cuestiones de eficiencia y fallos que hemos observado. Por último, se explicará de forma detallada la forma de añadir nuevos algoritmos a la aplicación.

Clases que intervienen en la ejecución del algoritmo.

A continuación se describirán con detalle las clases que intervienen en la ejecución del algoritmo, muchas de las cuales se han creado a partir de plantillas (templates), lo que quiere decir que son polimórficas (valen para cualquier tipo de datos). Las clases en cuestión son:

- **Clases creadas mediante plantillas:**

  - **Nodo:** Esta clase sirve para construir las listas doblemente enlazadas. Dichas listas se componen de éstos nodos, que se reservan de forma

dinámica y que contienen tres punteros: uno al nodo anterior, otro al nodo siguiente y por último un puntero al elemento en cuestión, que puede ser de cualquier tipo, incluyendo tipos dinámicos como listas, matrices, etcétera. La única restricción a éstos tipos es que tienen que tener definido el operador de asignación.

Las funcionalidades de los nodos son muy básicas, permitiendo obtener y modificar los punteros al nodo siguiente y al anterior. También se ofrece la posibilidad de devolver el puntero al elemento, así como de asignar un elemento nuevo, reemplazando al que estuviera. El parámetro de ésta última función es una dirección de memoria y no una copia del elemento con vistas a mejorar considerablemente la eficiencia, ya que si el elemento fuera una lista, al hacer la copia que se pasa como parámetro tendríamos un coste en tiempo y espacio de O(n), mientras que de ésta forma, el coste en ambas magnitudes es de O(1), es decir, constante. Éste criterio es el que se sigue también en la segunda constructora, la que no es por defecto.

- **Lista:** Como ya explicamos anteriormente, los elementos de ésta clase están contenidos en los nodos. La clase en sí, consta de tan sólo de tres punteros, así como de dos enteros. Los punteros apuntan al primer nodo de la lista, al último y al actual, es decir, al último nodo en el que se ha hecho algo, bien sea crearlo o bien sea posicionarnos en él. Los dos enteros hacen referencia a la longitud de la lista, así como a la posición del nodo al que apunta actual.

  Las funcionalidades de la lista son más numerosas. Permiten lo siguiente: crear una lista vacía o bien crearla copiando otra lista, es decir, una constructora de copia. También se ha implementado el operador de asignación. Por supuesto, se dispone de una destructora que libera la memoria de la lista adecuadamente cuando ésta se destruye. Asimismo, se incluyen métodos para saber si la lista está vacía o no, así como su longitud o la posición a la que apunta actual.

  Otras operaciones que se pueden realizar con lista son: añadir o borrar un elemento, que trabajan sobre el nodo actual, mover el puntero al nodo actual hasta la última posición (también se puede hacer con la primera), avanzar o retroceder actual nodo a nodo, obtener el puntero a los elementos a los que apuntan los punteros al nodo actual, primero y último, así como sustituir los valores de dichos elementos por otros nuevos. En dichas sustituciones, así como al añadir, se pasa un puntero al elemento como parámetro y no el elemento en sí. Esto es así por cuestiones de eficiencia. Es exactamente la misma situación que hemos descrito en los nodos anteriormente.

  Por último, se oferta la posibilidad de obtener un puntero al elemento de cualquier nodo de la lista, localizándolo por su posición y sin mover el puntero al nodo actual. También es posible mover el puntero al nodo actual a cualquier posición de la lista, así como eliminar todos los nodos de la lista.

Ya solamente restaría mencionar tres métodos que son protegidos y no públicos como los anteriores. Sirven para obtener los punteros a los nodos actual, primero y último y no a sus elementos. Son necesarios para construir otros métodos que hemos mencionado anteriormente, como la constructora de copia o el operador de asignación. Al ser protegidos, nos aseguramos de que sólo los utilicen otras listas, con lo que se garantiza que ningún tipo de datos que no sea una lista puede moverse por la estructura usando los métodos de la clase nodo. De ésta forma se obliga a usar los métodos de la lista apropiados.

- **Matriz2D:** Esta clase es, básicamente, un array bidimensional de dimensiones arbitrarias. Se compone de dos enteros, que almacenan el número de filas y columnas que posee dicho array. También está compuesta por un puntero a los elementos, que servirá para reservar la cantidad de memoria que sea necesaria de forma dinámica.

  Esta clase oferta las siguientes funcionalidades: una constructora de copia, una destructora, el operador de asignación, una constructora por defecto, así como otra en la que se puede indicar el número de filas y columnas, una función que permite obtener un elemento y otra que devuelve un puntero a ése elemento, en vez del elemento en sí. Esto último es más eficiente y es lo que se hace en otras partes del proyecto.

  Además, ésta clase permite conocer el número de filas y columnas que tiene, así como resetear la matriz, es decir, volver a inicializar todos sus elementos con su constructora por defecto. También se ha añadido la posibilidad de modificar el tamaño de la matriz de forma dinámica.

- **Matriz3D:** Ésta clase es prácticamente igual a Matriz2D solo que el array es tridimensional. Las funcionalidades son prácticamente las mismas. No se usa en ninguna parte del proyecto. Se creó con vistas al Tetris3D, cuya clase Ficha tendría una componente de ésta clase.

- **Clases que no están hechas mediante plantillas:**

  - **Tarea:** Esta clase representa una tarea. Sus componentes son: ancho, largo, tiempo de llegada, tiempo de ejecución, tiempo máximo, color, posición x, posición y baja, posición y alta y posición z. Color es un array de tres elementos (rojo, verde y azul).

    Aparte de los típicos métodos de acceso y actualización de los componentes, se han implementado las siguientes funciones: una constructora por defecto y otra que tiene como parámetro una cadena. Dicha cadena se analiza, convirtiendo los caracteres a números, con los que más tarde se inicializarán las componentes.

Ésta constructora se usa a la hora de cargar un histórico, del que se leen las líneas del fichero, convirtiendo las que correspondan en tareas que serán añadidas a una lista.

La última constructora es aquella en la que pasamos como parámetro las componentes importantes, inicializando a –1 las posiciones de la tarea, lo que puede ser útil para que el algoritmo (o el usuario) sepa si la lista de tareas está tratada o no.

Hay dos métodos toString: uno recortado y otro sin recortar. El recortado se usa para guardar las tareas en un fichero. Se limita a convertir a caracteres las componentes esenciales de la tarea, lo que no incluye el color ni las posiciones. El que no está recortado se usa para guardar un histórico y convierte todas las componentes a caracteres.

También hay dos funciones que permiten generar rangos de forma aleatoria, así como un método que hace que una tarea se pinte. Entre los parámetros de ésta última función, cabe destacar el booleano, que permite distinguir si queremos que se vean todas las tareas o sólo aquellas que están en ejecución y el parámetro tiempo. Éste último hace que las tareas se pinten más arriba o más abajo de lo que indica su posición y baja (la y alta es la posición y baja más el tiempo de ejecución). De ésta forma, se consigue simular el paso del tiempo sin alterar las posiciones, lo que haría que el histórico fuera incorrecto si se guardase después de hacer una simulación.

- **Tareas:** Es, básicamente, una lista de tareas. Tiene un componente adicional: la escala. Sirve para alargar adecuadamente el eje del tiempo, en función de los deseos del usuario. Como no iban a ser necesarias todas las funcionalidades de las listas, ésta clase se ha limitado un poco.

  Sus métodos son: constructora por defecto, añadir una tarea (siendo su parámetro un puntero), modificar la escala, ir al principio de la lista, ir a una posición cualquiera, devolver la posición en la que se encuentra el puntero actual de la lista, avanzar una posición, borrar todos los elementos, con lo que se obtiene una lista vacía y devolver la longitud de la lista, así como también puede obtenerse un puntero a la tarea actual o bien un puntero a una tarea cualquiera identificada por su posición.

  Otro método interesante es ponAMenosUno, que recorre la lista inicializando las posiciones de las tareas a –1. A pesar de que no se usa en ninguna parte del proyecto, se oferta por si algún algoritmo quiere hacer uso de ésta funcionalidad cuando empiece a ejecutarse. De ésta forma se garantizaría que las posiciones de las tareas están a –1 aunque se haya ejecutado otro algoritmo anteriormente. Así, no sería necesario grabar la lista de tareas y volver a cargarla para reinicializar sus componentes.

Queda un método público, que llama a otros dos (los únicos que son privados). Uno de ellos pinta la línea de tiempo y el otro pinta la lista de tareas.

- **Eventos:** Ésta clase hace el papel de lista de todos los eventos que pueden suceder. Se compone de un puntero a enteros, que se usará para reservar memoria para la lista de eventos en forma de array, así como de tres enteros más, que llevan la cuenta del número de eventos que hay en la lista, así como de qué eventos han sido tratados por el algoritmo y en qué evento está la simulación.

  Dentro de la lista, los eventos se representan como enteros, haciendo referencia al tiempo en que suceden. Se han tomado como eventos el tiempo de llegada de las tareas y el tiempo de llegada más el tiempo de ejecución.

  Las funcionalidades de los eventos son las siguientes: constructora por defecto, una función que borra todos los eventos de la lista y otra que devuelve la longitud de la misma. Los demás métodos permiten devolver la posición del puntero al tiempo del algoritmo, la posición del puntero al tiempo de simulación, así como el intervalo de tiempo por los dos lados. Por la derecha se devuelve el mínimo tiempo en el que ya han sucedido todos los eventos y por la izquierda se devuelve el máximo tiempo en el que no ha sucedido ningún evento.

  También se puede tener acceso al tiempo en el que ocurre un evento determinado por su posición (con la consiguiente comprobación de rangos del array), así como actualizar los dos punteros de tiempo adecuadamente. Esto último es necesario debido a que puede haber varios eventos que sucedan a la vez. Como no pueden suceder algunos sí y otros no, cuando se pide avanzar tres eventos, por ejemplo, lo que se hace en realidad es avanzar al menos tres eventos y después se actualiza el puntero correspondiente de forma coherente (hacia delante en el caso del algoritmo o en cualquiera de las dos direcciones en el caso de la simulación).

  La función de reasignación de memoria (resize) reserva un array del número de elementos que se le pida (siempre y cuando sea positivo) más uno, destruyendo el array antiguo si existiese. El último elemento sirve para saber si hemos tratado todos los eventos, ya que los punteros se posicionan en el siguiente evento sin tratar. Si el puntero está en el último elemento es que hemos tratado todos los eventos, mientras que si está en el primero no hemos tratado ninguno.

  El último método crea una lista de eventos en función de una lista de tareas. Dicha lista ya está ordenada de menor a mayor mediante un algoritmo de ordenación rápida. La complejidad de éste algoritmo es de O(n log n) en el caso promedio, aunque es cuadrática en el caso peor.

- **Intervalo:** Esta clase representa un intervalo de alturas de una posición concreta de la FPGA. Las componentes que lo forman son: altura mínima, altura máxima, si está ocupado o libre y el ancho. Éste último campo se ha introducido por cuestiones de optimización del algoritmo. Si se llega a una posición ocupada, la siguiente posición que se analizará será la que esté a ancho de distancia, ya que las de en medio estarán ocupadas también.

  Las funcionalidades de ésta clase son muy básicas. Son métodos de acceso y modificación de las componentes, así como un par de constructoras, una de ellas por defecto.

- **NodoMatriz:** Esta clase representa un elemento (posición) de la FPGA. Es, básicamente, una lista de intervalos de alturas. De hecho, sólo se compone de dicha lista.

  Como no eran necesarias todas las funcionalidades de las listas, en ésta clase se han recortado un poco sus capacidades. Sólo cabe destacar que siempre hay al menos un elemento en las listas.

- **FPGA:** Ésta clase se compone de tres elementos: el comentario asociado, el nombre y una Matriz2D de elementos NodoMatriz. Sobre éste último componente trabajan los algoritmos.

  A parte de la constructora y los métodos de acceso y modificación de componentes, cabe destacar el reset, que reinicializa todos los nodos de la matriz, así como el método para que la FPGA se pinte y la funcionalidad de convertirse a cadena (toString). Se hace uso de éste último método en el histórico.

- **Algoritmo:** Ésta clase consta de unos cuantos métodos públicos. Sirve para definir los algoritmos a usar. Aquí es donde se han de añadirse los algoritmos.

  Hay una variable que lleva la cuenta del número de algoritmos que hay definidos. Cada vez que se añada un algoritmo, debería incrementarse ésta variable a mano.

  Ésta clase posee una constructora por defecto y un método que devuelve el número de algoritmos definidos. Hay dos métodos más: uno que devuelve el nombre que se le da a cada algoritmo, identificándolo por su número y otro que manda ejecutar un rango de tiempos a un algoritmo dado, identificado por su número, pasándole como parámetros una FPGA, una lista de tareas y un comentario en el que el algoritmo podrá introducir mensajes que se visualizarán posteriormente.

- **Interfaz:** En ésta clase se centraliza el manejo de casi toda la aplicación. Tiene unos cuantos componentes esenciales desde el punto de vista del algoritmo. Dichos componentes son: un objeto de la clase algoritmo, dos variables que llevan la cuenta de los tiempos (de ejecución y simulación),

el comentario histórico, que es donde se almacenan los comentarios hechos por el algoritmo, la FPGA, la lista de tareas, un booleano que indica si el algoritmo ha terminado de procesar todas las tareas (su nombre es por_terminar), la lista de eventos y una variable que indica el número del algoritmo que se está ejecutando.

Otros componentes de interés son el modo (on-line u off-line) y el número de pasos o eventos que se manda hacer al algoritmo (o a la simulación). Se espera que si alguno de los dos (pasos o eventos) es distinto de cero, el otro sí que debe ser cero.

De entre sus numerosos métodos, el más interesante en cuanto a lo que el algoritmo se refiere es ejecutar. Ejecutar tiene en cuenta el estado actual de variables como por_terminar, número de algoritmo, modo, número de pasos o eventos y en función de ellas manda ejecutarse al algoritmo actual, actualizando posteriormente las variables que sean necesarias.

Otros métodos de interés son los que garantizan que se cumplan ciertas precondiciones, como resetea, reseteaTareas, creaEventos, etcétera. Dichos métodos serán llamados desde la interfaz gráfica (GUI) cuando sea pertinente, como antes de hacer una llamada a ejecutar, por ejemplo.

Algoritmo de ejemplo:Prueba

Esta sección está dedicada a nuestro algoritmo, cuyo nombre es Prueba. Está pensado para ejecutarse en modo off-line. De hecho, aunque se ejecute en modo on-line, trata toda la lista de tareas. Es responsabilidad del algoritmo tratar de forma adecuada el rango de tiempos que se le pasa como parámetro. Tampoco se preocupa de los tiempos máximos de las tareas.

En cuanto a los comentarios que Prueba devuelve a la aplicación para que sean visualizados en el campo memo cuyo título es histórico, se limitan a indicar en qué posición (en coordenadas cartesianas x, y, z) se ha ubicado cada tarea. Esto nos fue de muchísima utilidad a la hora de depurar, así que se lo recomendamos a cualquiera que tenga que hacer algoritmos después. Al fin y al cabo, para cambiar los comentarios dentro del algoritmo siempre hay tiempo.

Queda un último detalle por señalar en cuanto a los sistemas de coordenadas: en el sistema de coordenadas de lo que se muestra en el campo memo, se toma la z como la altura, mientras que a la hora de pintar es el eje y el que se toma como altura, de ahí lo de y baja e y alta en las tareas. Lo único que tuvimos que hacer fue una pequeña traducción de unas coordenadas a otras a la hora de cambiar las posiciones de las tareas, que se rigen por el sistema de coordenadas que usa el eje y como altura.

La idea de hacer un algoritmo surgió de la necesidad de tener una visualización lo más compacta posible y sin solapamiento de tareas, cosa que seguramente no se hubiera conseguido colocando las tareas de forma aleatoria, aun comprobando si las

posiciones habían sido ocupadas ya, como era nuestra primera aproximación. Al estar distribuidas de forma compacta las tareas, nos fue mucho más fácil hacernos una idea de cómo iban a verse después de ser colocadas según un criterio más o menos eficiente. De ésta forma se pueden pintar más grandes y se ven mucho mejor. Este algoritmo sirve, además, para que los que tengan que implementar nuevos algoritmos después tengan un ejemplo de partida sobre el que trabajar.

Prueba sigue un criterio de colocación first-fit. Básicamente, lo que hace es recorrer la lista de tareas, intentando colocar cada tarea en el primer sitio libre y suficientemente grande que encuentre. A partir de la primera tarea colocada, se tienen que volver a poner en la primera posición todas las listas de intervalos de altura de la matriz de la FPGA. Esto es así porque la siguiente tarea puede ser más pequeña que la actual y caber en un hueco en la que ésta no cabría, es decir, hay que volver a analizar todos los huecos que hemos tratado anteriormente para intentar ubicar la siguiente tarea en alguno de ellos.

Una posible optimización, que no se ha implementado, sería no reinicializar las listas de intervalos de alturas si se sabe que la siguiente tarea tiene todas sus dimensiones mayores o iguales que la actual, con lo que es imposible que quepa en alguno de los huecos anteriores.

Se usan dos procedimientos auxiliares: uno para saber si una tarea se puede colocar en una posición y otro para que en caso de que así sea, se actualicen adecuadamente las listas de intervalos de altura poniendo esa región del espacio como ocupada. Ambos están situados en una clase que hemos dado en llamar Auxiliar.

Como las listas de intervalos de alturas tienen un puntero que señala al nodo actual (de ahí el hacer nosotros la clase lista y no usar las que vienen predefinidas en el lenguaje C++), el coste de los dos procedimientos auxiliares, llamados cabe y recubre es cuadrático y no cúbico, como cabría esperar.

Con ésta optimización se consigue reducir considerablemente el coste en tiempo del algoritmo, debido a que las llamadas a éstas funciones se encuentran ya de por sí muy anidadas dentro de varios bucles. Para que esto funcione correctamente, hay que asegurarse de que el puntero al nodo actual de todas las listas de intervalos de altura de la matriz de la FPGA esté posicionado en el intervalo de altura que contiene a la altura que se está tratando en cada momento concreto.

Hemos observado que el algoritmo presenta a veces un comportamiento anómalo. Si se ejecuta sobre listas de tareas muy grandes sobre FPGAs muy pequeñas (con lo que hay muchos niveles de alturas), coloca algunas tareas en alturas no óptimas según el criterio first-fit, es decir, no las coloca en el primer hueco libre, sino unas cuantas alturas por encima.

Esto puede deberse, casi con toda seguridad, a que los punteros a los nodos actuales de algunas listas de intervalos de alturas no están bien actualizados. La forma más fácil de solucionar esto es no optimizar, haciendo que el orden de complejidad de las funciones auxiliares sea cúbico. Para ello, dentro de dichas funciones habría que llevar el puntero actual de cada lista a su primera posición y recorrer la lista a partir de ahí hasta situar el puntero en el intervalo correspondiente a la altura que se está tratando.

Como añadir nuevos algoritmos a la aplicación.

Ésta aplicación está pensada para que sea muy sencillo introducir algoritmos nuevos. Sólo es necesario retocar un poco la clase algoritmo. La forma de llevar a cabo éste retoque se describirá en las líneas que siguen:

Lo primero de todo, como comentamos ya anteriormente cuando detallamos la clase algoritmo, es incrementar la variable numAlgoritmos, que se encuentra en el archivo de cabecera (.h) de dicha clase. Todo lo demás ha de añadirse en los procedimientos del archivo de implementación (.cpp).

Dichos procedimientos son: dameNombre y ejecuta. Ambos tienen algo en común: todo su cuerpo es un switch. Lo único que habrá que hacer es añadir una entrada a cada uno de los dos switchs por cada algoritmo que se quiera agregar.

En el caso de dameNombre, dicha entrada se limitará a devolver una cadena que represente el nombre del algoritmo, mientras que en el caso de ejecuta la entrada debe hacer una llamada al algoritmo en cuestión usando como parámetros los parámetros de la propia función ejecuta.

El algoritmo en cuestión puede escribirse como parte del archivo de implementación de la clase algoritmo, como es el caso de Prueba o bien puede escribirse en una unidad a parte añadiendo la consiguiente línea #include que incluya dicha unidad en el ya mencionado archivo de implementación.

Eso es todo en cuanto al agregado de algoritmos se refiere. Quedaría por señalar una serie de restricciones a las que han de someterse los algoritmos que funcionen bajo ésta aplicación. Son éstas cuestiones las que se van a detallar a continuación:

La aplicación sólo se hace responsable de mandar ejecutar al algoritmo un rango de tiempos razonable. Las variables que llevan la cuenta del tiempo en interfaz contienen la primera unidad de tiempo no tratada, es decir, en el tiempo cuatro, por ejemplo, sólo se han producido los eventos que vayan del tiempo cero (el tiempo inicial) al tiempo tres, ambos inclusive. Los eventos del tiempo cuatro están a punto de suceder, pero no han sucedido aún, de ahí que cuando se pinten las tareas, lo hagan una unidad de tiempo por detrás de la que estamos. En el ejemplo anterior se pintarían en el tiempo tres.

Es por esto que las tareas se pintan inicialmente en el tiempo –1. Al estar en el tiempo cero, en el que todavía no ha sucedido nada, ni siquiera los eventos del tiempo cero, sólo han sucedido los eventos hasta el tiempo –1. Cabe recordar aquí que el tiempo de simulación y el tiempo máximo tratado por el algoritmo pueden ser distintos, con lo que las tareas se pueden pintar en el tiempo –1 aunque el algoritmo las haya procesado todas.

El rango de tiempos que se le pasa al algoritmo va de la primera unidad de tiempo no tratada hasta la primera unidad de tiempo que no debe tratar, es decir, es un intervalo abierto por la derecha y cerrado por la izquierda. Por supuesto, todo lo comentado anteriormente es para el caso on-line. En el caso off-line se le manda

procesar un rango de tiempos que va del tiempo cero al tiempo en que se produce el último evento más uno, ya que si el último evento se produce en el instante treinta y tres, por ejemplo, hasta el tiempo treinta y cuatro no han sucedido todos los eventos.

Es responsabilidad del algoritmo tratar adecuadamente el rango de tiempos que se le pasa como parámetro (cosa que ya comentamos anteriormente en la sección del algoritmo prueba), como también lo es moverse por la lista de tareas hasta posicionarse en el nodo que le interese antes de seguir tratando dicha lista. Sólo se garantiza que la lista estará en su primera posición, así como que la FPGA estará reseteada y el comentario histórico estará vacío antes de mandar ejecutar el algoritmo en modo off-line.

Por supuesto, también es responsabilidad del algoritmo llevar la cuenta de qué tareas han sido tratadas y cuáles no. Esto puede hacerse llevando una lista interna de posiciones tratadas de la lista de tareas.

Hay otra forma de saber si una tarea ha sido tratada o no: sólo hay que analizar la posición x y la posición z. Ambas se inicializan a –1 cuando se crea la tarea, así que seguirán teniendo éste valor si no han sido tratadas o tendrán otro distinto si sí que lo han sido. Por esta razón, para conservar una lista de tareas y tratarla con un algoritmo distinto, habría que guardar la lista de tareas y volver a cargarla, porque si no, no se vuelven a reinicializar sus posiciones.

No debemos preocuparnos de cómo se pintan las tareas que no hayan sido tratadas cuando estemos en modo on-line. Sólo se pintarán aquellas tareas que hayan sido tratadas por el algoritmo, es decir, aquellas que tengan la posición x y la posición z distintas de –1 (realmente sólo se comprueba la posición x).

Tetris 3D

En ésta sección se describirán algunas de las ideas que teníamos para llevar a cabo la implementación de la parte más lúdica del proyecto: el tetris 3D.

Se implementó la clase Matriz3D con vistas a realizar ésta parte de la aplicación, como ya comentamos cuando introdujimos algunas de las clases que habíamos implementado. Se iba a usar ésta clase para realizar el tablero de juego, así como las fichas.

El tablero iba a ser una Matriz3D de punteros a fichas. Esta idea, en principio complicada, surgió por motivos de eficiencia. De ésta forma, la propia ficha se encargaría de comunicar al tablero la posición de la próxima casilla no ocupada por ella en la dirección dada, es decir, la próxima casilla a analizar, ahorrando varias comprobaciones. La idea es similar a la que hay detrás de la componente ancho de los intervalos de altura.

La clase ficha tendría los siguientes componentes: por un lado estaría el color y por otro la forma.

El color podría representarse como un número entero que indicase qué posición de una tabla de colores habría que usar para acceder a la cantidad de rojo, verde y azul a utilizar. Sería como una paleta de colores.

La forma nos dio mucho más que pensar. Cada ficha puede tener veinticuatro posiciones distintas en el espacio. Hay dos alternativas: una es tener una sola Matriz3D de booleanos suficientemente grande como para albergar a la ficha en cualquier posición (una matriz de 4 x 4 x 4, por ejemplo), donde los booleanos indiquen qué posiciones están libres y cuáles no. La otra sería tener veinticuatro matrices 3D diferentes, una por cada posición. Ambas tienen ventajas e inconvenientes, que se detallarán a continuación:

La primera opción supone un ahorro en cuanto a espacio. Tiene la ventaja añadida de que la operación de crear fichas nuevas en tiempo de ejecución es muy sencilla. Por el contrario, algo tan común como rotar la pieza con respecto a cada uno de los tres ejes introduce retardos temporales cúbicos, ya que hay que recolocar todos los booleanos de la Matriz3D.

La otra opción no supone costes de recolocación de la matriz cada vez que la pieza sufre una rotación, aunque conlleva un considerable coste en espacio. De todas formas, el coste espacial es mucho menos importante que el coste temporal. El verdadero problema de ésta opción viene de lo complicado (y costoso en tiempo) de calcular las veinticuatro posiciones posibles de una pieza que se cree en tiempo de ejecución.

Si a esto se le añade el problema de que el número posible de piezas distintas puede aumentar considerablemente cuando se consiga hacer un suelo (versión 3D de la línea), que partiría ciertas piezas haciendo que parte desapareciese y parte siguiese en el

tablero, convirtiéndose en una ficha nueva, se ve claramente que no se pueden precalcular en tiempo de compilación.

Se tenía pensado hacer una serie de optimizaciones para que las comprobaciones espaciales que hay que hacer durante el juego fuesen más eficientes.

Por un lado, se quería hacer uso de una Matriz2D de alturas máximas, que sirviese como apoyo a la hora de comprobar si las piezas podían moverse en cualquiera de los tres ejes. Así, el coste de las comprobaciones se reduciría drásticamente al eliminar la necesidad de comprobar una de la dimensiones.

También se quería llevar la cuenta de la altura más baja libre de dicha Matriz2D, así como de la altura más alta ocupada de la misma. Ambos valores se podrían almacenar en dos variables enteras que ahorrasen ciertas comprobaciones, como por ejemplo, si la ficha estuviese más alta que la máxima altura ocupada, podría caer sin problemas, sin necesidad de ningún tipo de comprobación. Por otro lado, comprobar alturas por debajo de la mínima altura disponible es absurdo, con lo que ambas variables acotan el rango de alturas a comprobar.

Más tarde, nos dimos cuenta de que la Matriz2D no era de utilidad, debido a que es posible colocar una pieza por debajo de la altura máxima ocupada, suponiendo que debajo haya algunas casillas libres.

Todas estas ideas surgieron antes de empezar con la implementación de la otra parte del proyecto. Con la perspectiva que tenemos ahora, nos damos cuenta de que aunque la representación de fichas como matrices 3D es buena, quizá sería mejor representar el tablero como una Matriz2D de listas de intervalos de altura, como se hizo con las FPGAs.

Por supuesto, también tomamos en consideración la representación propuesta por nuestro tutor, Julio Septién, que sugería representar el tablero como una Matriz3D de casillas, que contendrían un campo booleano para saber si la casilla está libre u ocupada, así como un campo para el color, de forma que cuando una ficha fuera colocada, dejase de existir como tal, actualizando las posiciones ocupadas del tablero, así como los colores.

5. APÉNDICES.

A.1. Manual de usuario

A.2. Bibliografía

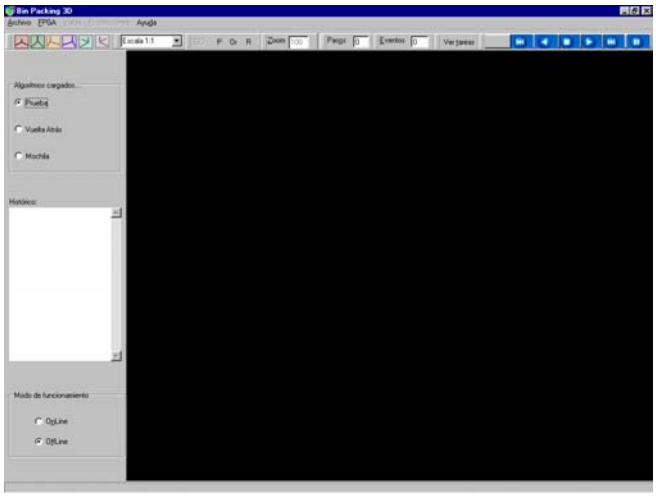
A.3. Autorización

A.2. Glosario

A.1 MANUAL DE USUARIO

La aplicación representa un entorno de ventana en el cual interactuar con el programa resulta sencillo e intuitivo. La mayoría de las funcionalidades se consiguen tanto por desplazamiento entre los distintos menús como por los botones de acceso rápido situados en la zona superior a la ventana de representación, ofreciéndonos la posibilidad de solicitar acciones tanto por teclado como desde ratón.

En el momento en el que lanzamos la aplicación se abre ante nosotros la ventana principal,

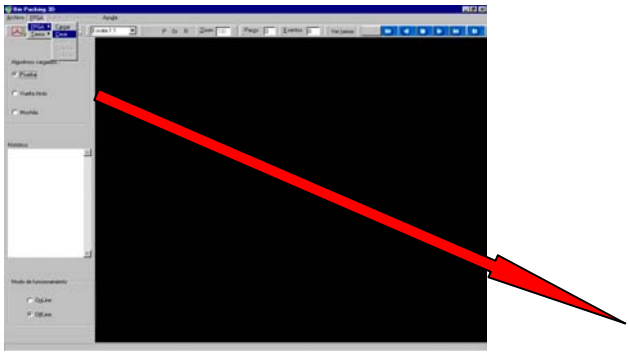


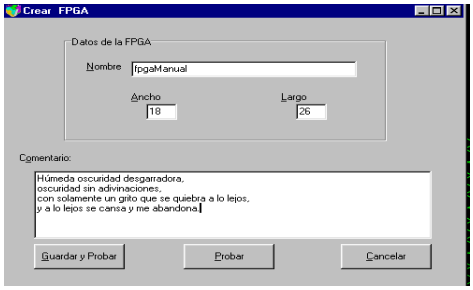
El recuadro negro albergará la representación de la simulación mientras que el espacio blanco identificado como histórico mostrará mensajes de texto informativos para comunicarnos las distintas situaciones en las que el programa se encuentre.

Ahora podemos cargar una representación o bien generar una, para lo cual tendremos que elegir una FPGA, una lista de tareas y un algoritmo de los que se muestran en la parte superior izquierda de la pantalla, el menú **Ayuda->Ayuda** nos indica las pautas para una correcta ejecución, pero veamos las distintas posibilidades que nos ofrece el programa:

1. Crear una nueva FPGA

Para trabajar con una determinada FPGA tenemos dos opciones, cargarla desde fichero o bien crearla nosotros mismos en ejecución; esto lo conseguimos mediante el menú **FPGA->FPGA->Crear** que nos conducirá a una nueva ventana donde se solicitan los datos que la caracterizaran.





Debemos incluir un nombre que no contenga espacios en blanco puesto que de ser así se ignoraría todo lo que estuviera a continuación del primero de ellos.

Los campos ancho y largo deben ser rellenados por enteros comprendidos entre los valores 2 y 100, ambos incluidos, y el comentario puede ser rellenado o no, en cualquier caso si queremos utilizar en la ejecución actual la FPGA que acabamos de definir, debemos salir de la ventana escogiendo **Probar** o bien **Guardar y Probar**, en caso contrario, elegiremos **Cancelar.**

Las dos primeras opciones fijan en memoria los datos introducidos con la diferencia de que **Guardar y Probar** nos mostrará un cuadro de diálogo para grabar la FPGA en un fichero de nuestra elección ya existente o bien nuevo. Si el fichero fuese nuevo, tan sólo tendríamos que escribir su nombre en el campo "Nombre de archivo" (la extensión asociada se respetará en todo momento, sin embargo, si no escribimos extensión, automáticamente se generará .TXT)

los datos se grabarán en el fichero indicado, al final del mismo y respetando todas las líneas anteriores.

Si no estamos seguros de querer añadir está FPGA a ningún fichero, podemos usarla en esta ejecución y luego solicitar grabarla marcando: **FPGA->FPGA->Guardar**

Otra manera de generar una nueva FPGA es escribirla directamente en el fichero de FPGAs desde cualquier editor de texto. Para ello deberemos seguir las restricciones

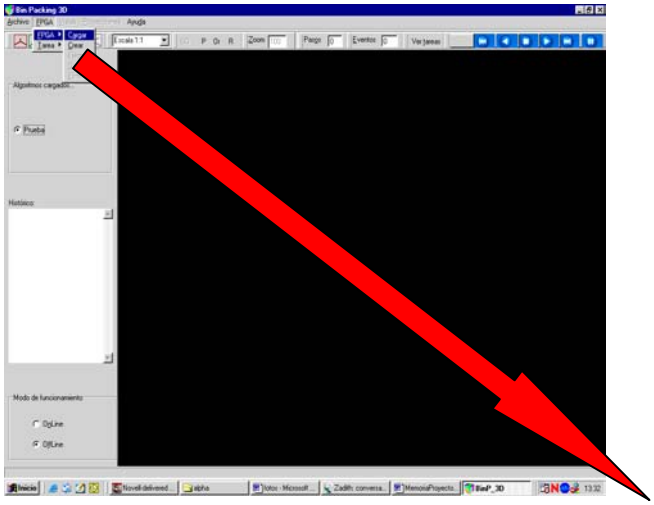
ya comentadas además de empezar cada línea de comentario por un *, el comentario debe ir inmediatamente después de la línea destinada a la FPGA, línea compuesta por NOMBRE ANCHO LARGO:

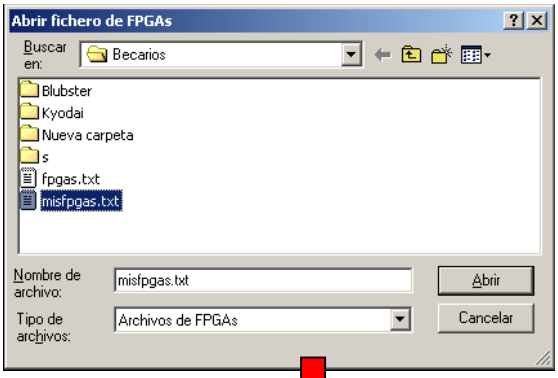
```
FPGAinventada 18 24
*Pero no dice nada, no las suelta.
*Entonces miro en lo oscuro llorando,
*y me envuelvo otra vez en mi noche
*como en una cortina pegajosa
*que nadie nunca nadie nunca corre.
```

Una vez tenemos un fichero de FPGAs podemos cargarlo y dispondremos de la opción de elegir cualquiera de ellas para nuestra ejecución:

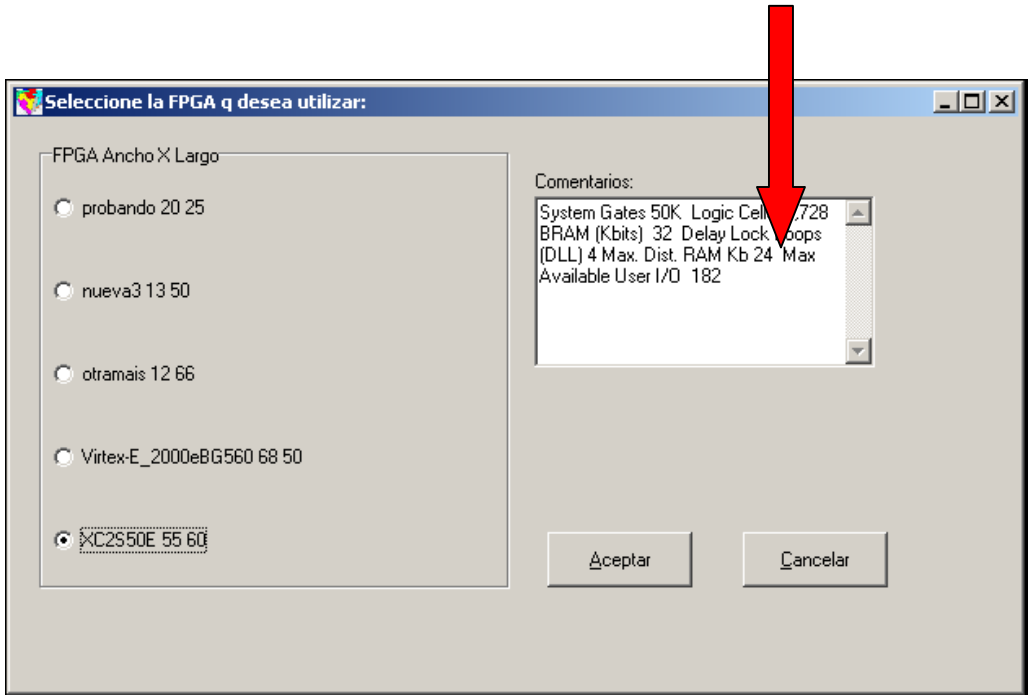
2. Cargar FPGA

Solicitando desde menú **FPGA->FPGA->Cargar** se abrirá ante nosotros un cuadro de elección de ficheros que nos permitirá escoger cada una de las FPGAs que contenga:



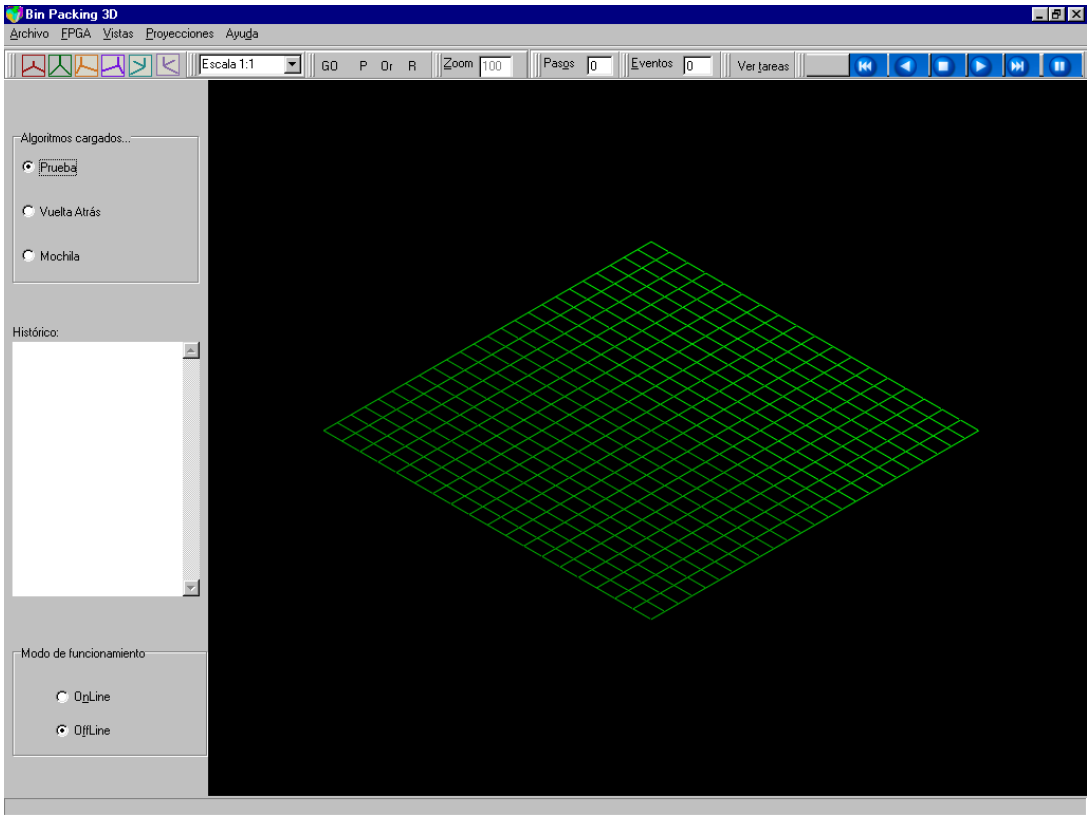


Si aceptamos abrir el archivo, iremos directamente a la ventana de elección de FPGA, ventana que también se alcanza desde el menú **FPGA->FPGA->Elegir**:



Ahora procedemos a elegir una de las FPGAs que se ofertan de cara a su posterior uso en la ejecución. Si pulsamos cancelar en este formulario, ninguna de ellas será cargada, sin embargo, se guarda una copia de todas en memoria de manera que no hay porqué volver a cargar el fichero, seleccionando **FPGA->Elegir** cada vez que queramos una nueva.

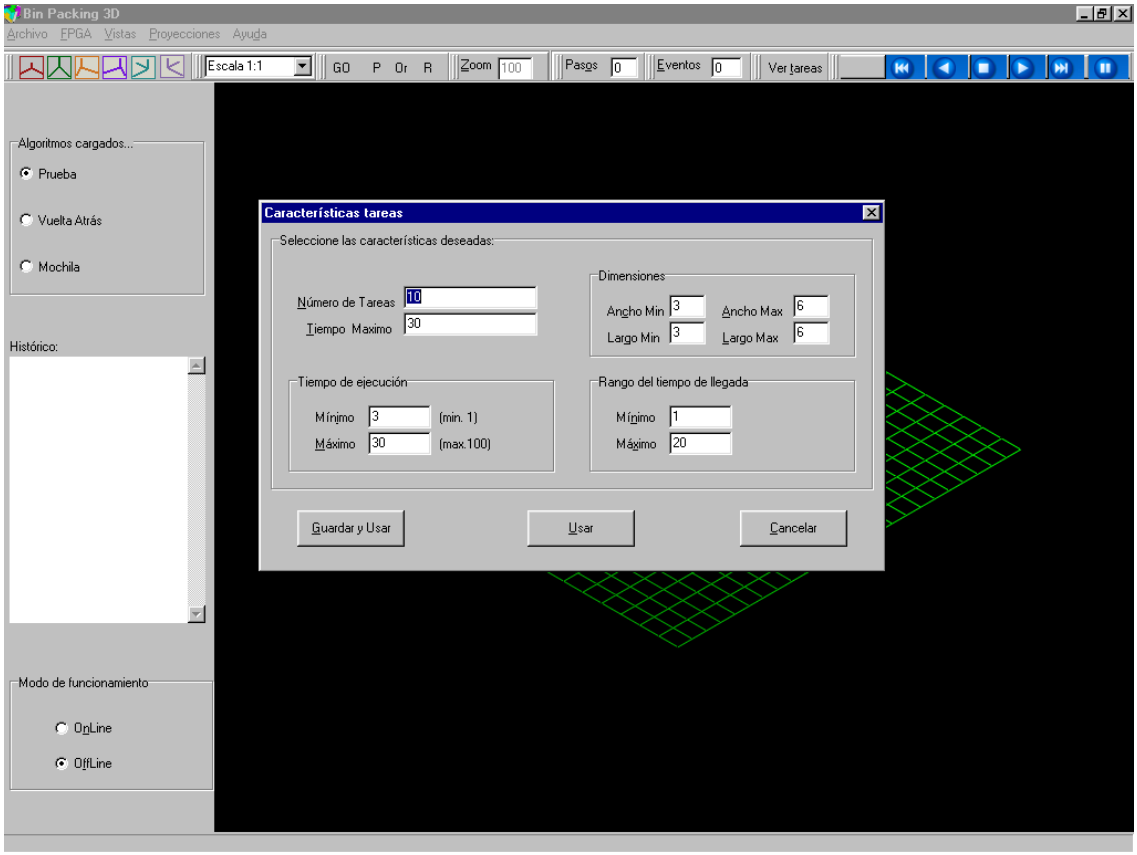
Si pulsamos aceptar, tenemos una FPGA sobre la que aplicar el algoritmo:



Lógicamente, antes de ejecutar ningún algoritmo se debe cargar una lista de tareas hardware:

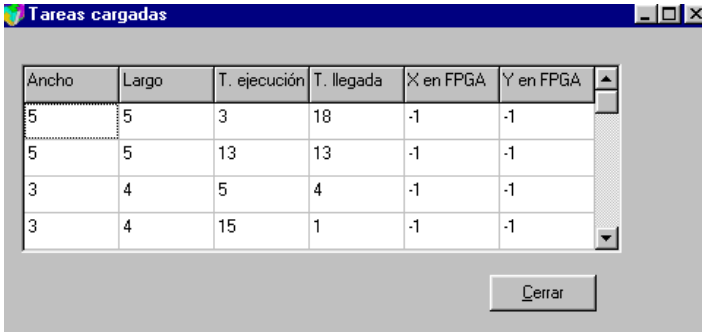
3. Crear un nueva lista de tareas

Al igual que las FPGAs, las tareas pueden crearse en ejecución o bien cargarse desde un fichero, si deseamos crearlas debemos elegir el menú **FPGA->Tareas->Crear**, lo que nos llevará a la pantalla de elección de límites para la lista (las características de las tareas se generan aleatoriamente entre los rangos seleccionados). Por una mayor comodidad, los valores aparecen rellenos por defecto, pudiendo el usuario modificar aquellos que desee, debemos recordar que podemos cambiar de un campo a otro utilizando el tabulador, que seguirá un orden lógico de izquierda a derecha y de arriba abajo o mediante la combinación de Alt+X (siendo X el carácter subrayado de la casilla en la que nos queremos situar).



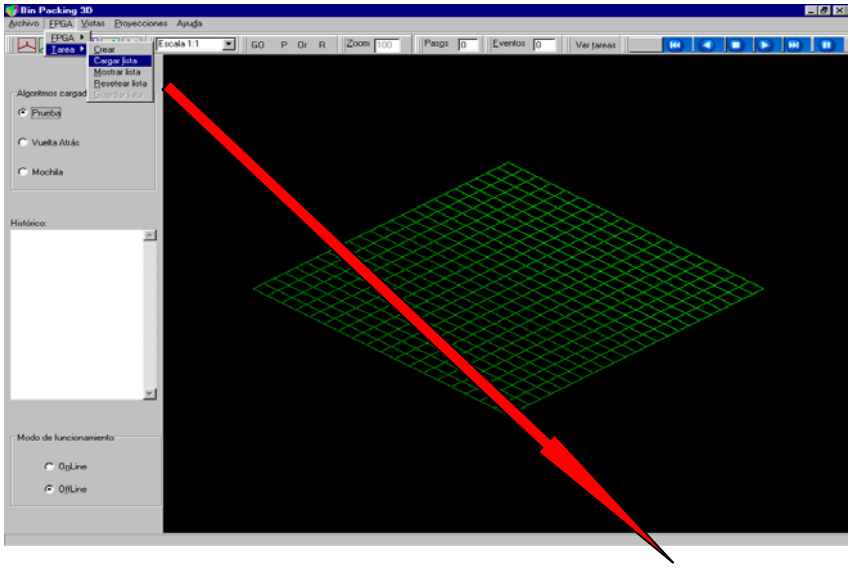
Al igual que en las FPGAs, podemos elegir utilizar directamente la lista o grabarla primero, la grabación se llevará a cabo bien en un fichero de tareas (extensión TSK) cuyo nombre proporcionaremos o bien en cualquier otro a nuestro gusto.

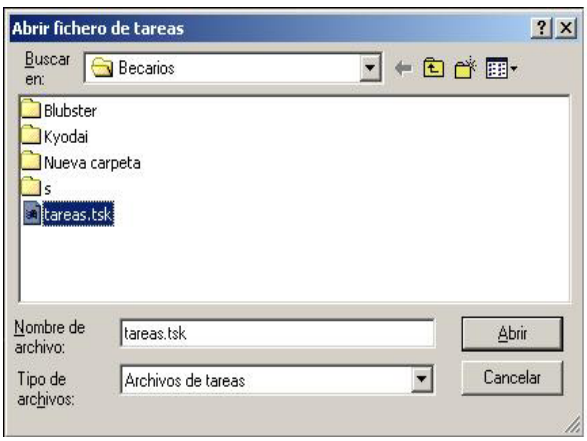
Una vez la lista esté cargada, podemos comprobar sus características en el menú **FPGA->Tareas->Mostrar** lista que nos sacará por pantalla la lista con sus posiciones actuales, si aun no ha sido tratada, tendrá prefijadas a –1 todas sus coordenadas en la FPGA mientras que si el algoritmo ya ha "empaquetado" las tareas podremos ver que lugar ha asignado a cada una de ellas en modo texto:



4.Cargar una lista de tareas Hardware

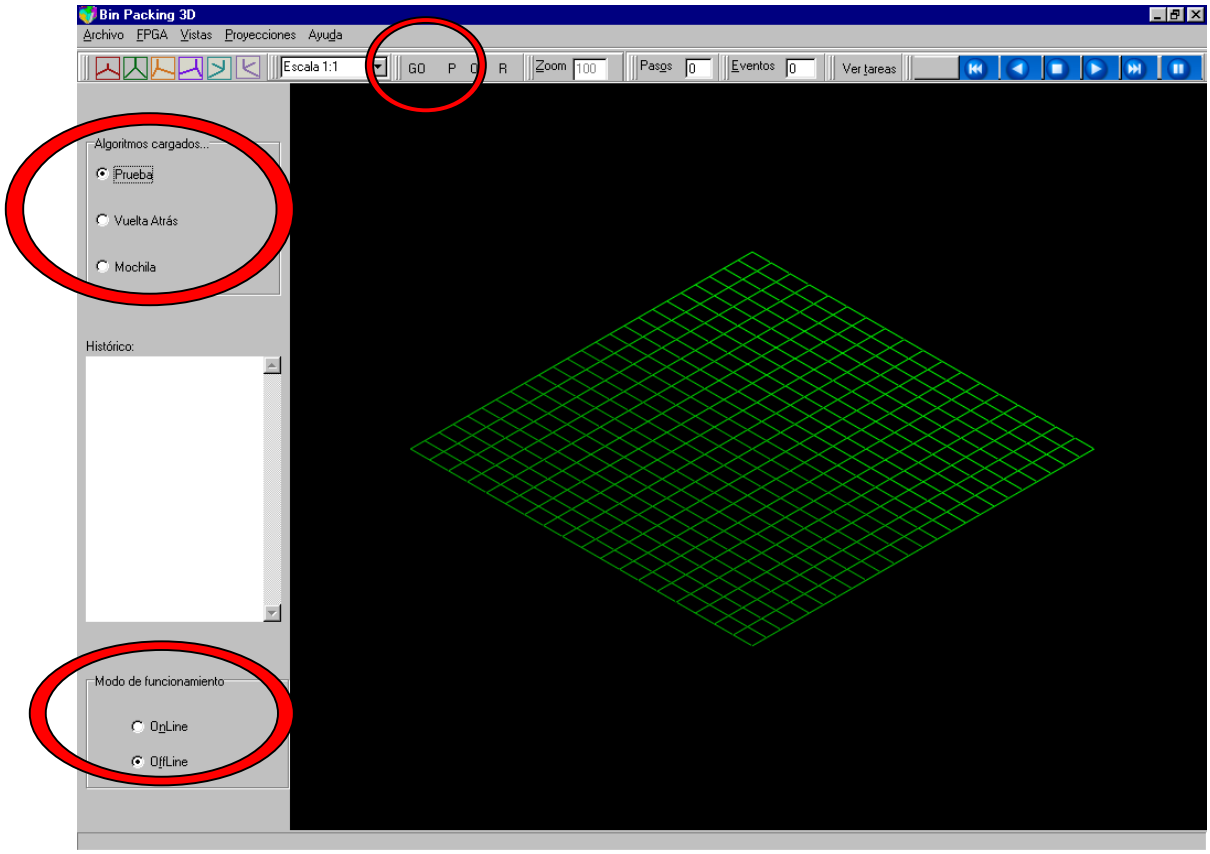
Si hemos guardado alguna lista, podemos recuperarla mediante el menú **FPGA->Tareas->Cargar** lista que mostrará los archivos de tareas que tenemos. A diferencia de las FPGAs, cada lista de tareas está almacenada en un fichero independiente que no conviene editar, los datos almacenados en ese fichero se corresponden con la creación de tareas sin tratar, esto es las posiciones que ocuparán en la FPGA no son almacenadas.





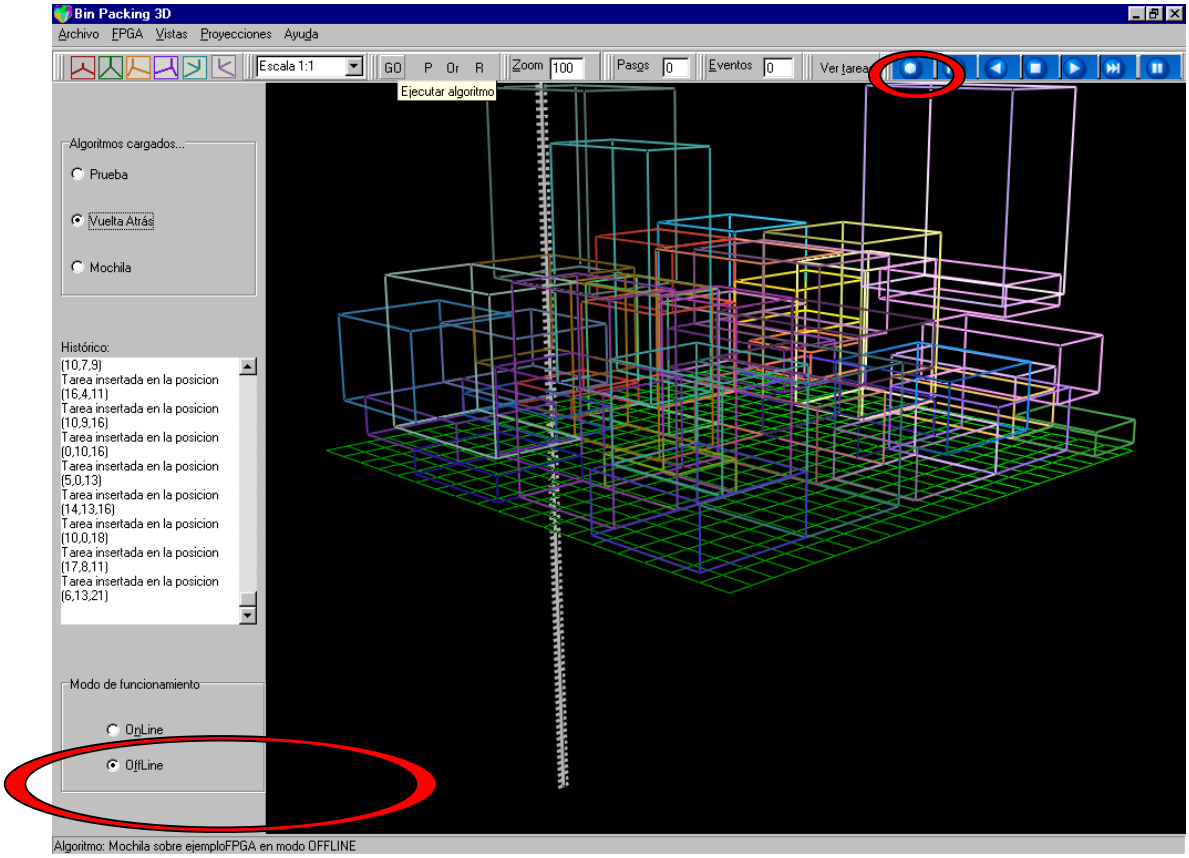
5. Ejecución

Una vez cargada tanto la lista como la FPGA, podemos ejecutar el algoritmo que deseemos pinchando en el botón de GO. El algoritmo a utilizar se selecciona pinchando en su nombre, a la izquierda de la pantalla (por defecto, está señalado el primero definido).



El modo de ejecución, por defecto OffLine, ha de estar señalado a nuestro gusto antes de ejecutar, lo que podemos hacer desde el menú **Archivo->Simulación** o bien marcando en la esquina inferior izquierda nuestra elección.

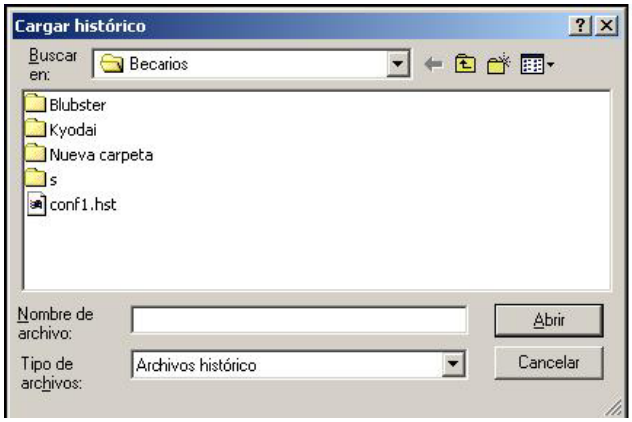
Una vez procesadas las tareas, se mostrarán en el área destinada a tal efecto, pudiendo ser controlada la simulación a nuestro antojo (ver simulación). En la parte inferior de la pantalla aparecerá una línea de texto indicando el algoritmo utilizado y sobre qué FPGA se ha realizado, además del modo en el que se procesó. Si lo deseamos, podemos grabar el resultado de aplicar el algoritmo deseado sobre la FPGA elegida y las tareas cargadas en un archivo histórico: **Archivo->Guardar** configuración, o bien en el botón record situado en la zona superior derecha:

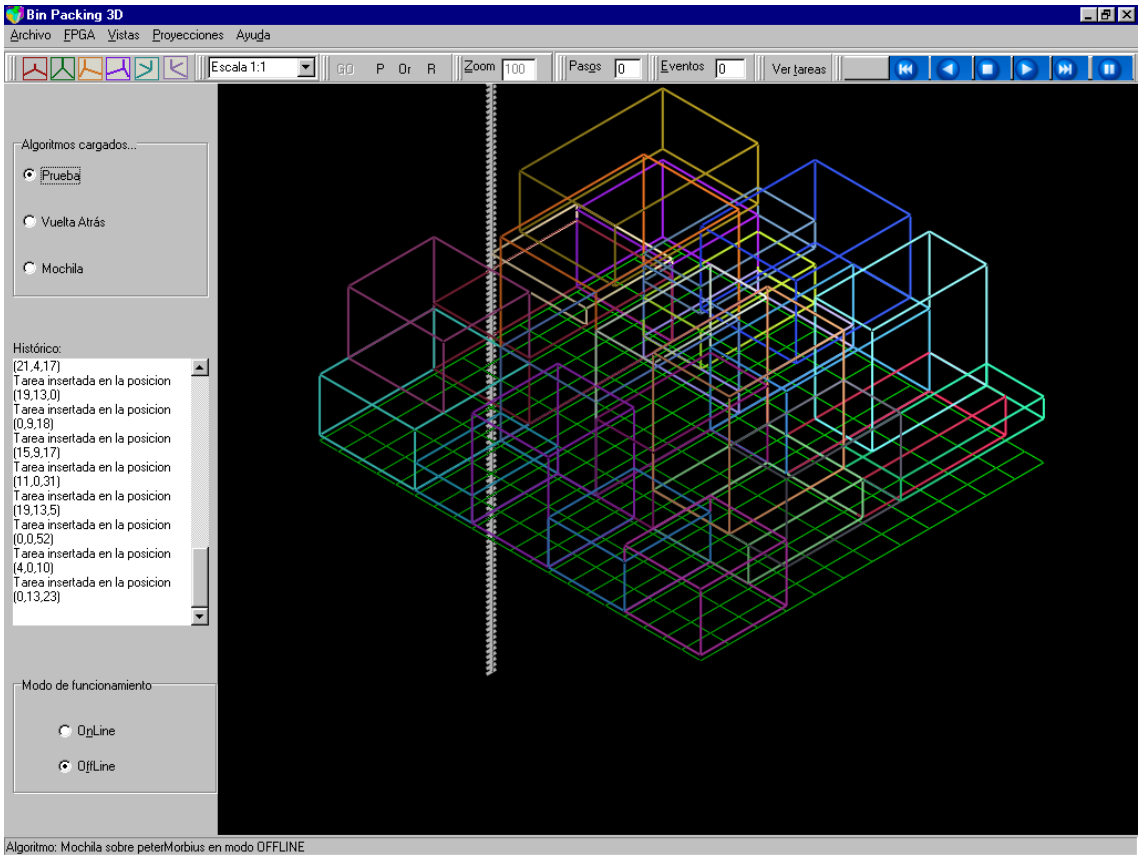


La grabación se generará por defecto en un archivo de extensión HST, que luego podremos cargar para visualizar los resultados sin necesidad de volver a escoger todos los parámetros:

6. Carga de un histórico

Mediante el menú **Archivo->Abrir** configuración llegaremos a un ya conocido cuadro de diálogo donde se nos muestran los ficheros a cargar, una vez seleccionado el deseado veremos una pantalla similar a la anterior (cuando el algoritmo se había ejecutado) salvo por el botón de grabación, que aparece desactivado: si el mismo histórico queremos guardarlo con dos nombres distintos deberemos hacerlo al probarlo (dos veces) o bien desde el sistema, cambiando directamente el nombre del archivo.





Una vez todas las tareas están ubicadas, de una manera u otra, podemos pasar a la:

7. Simulación

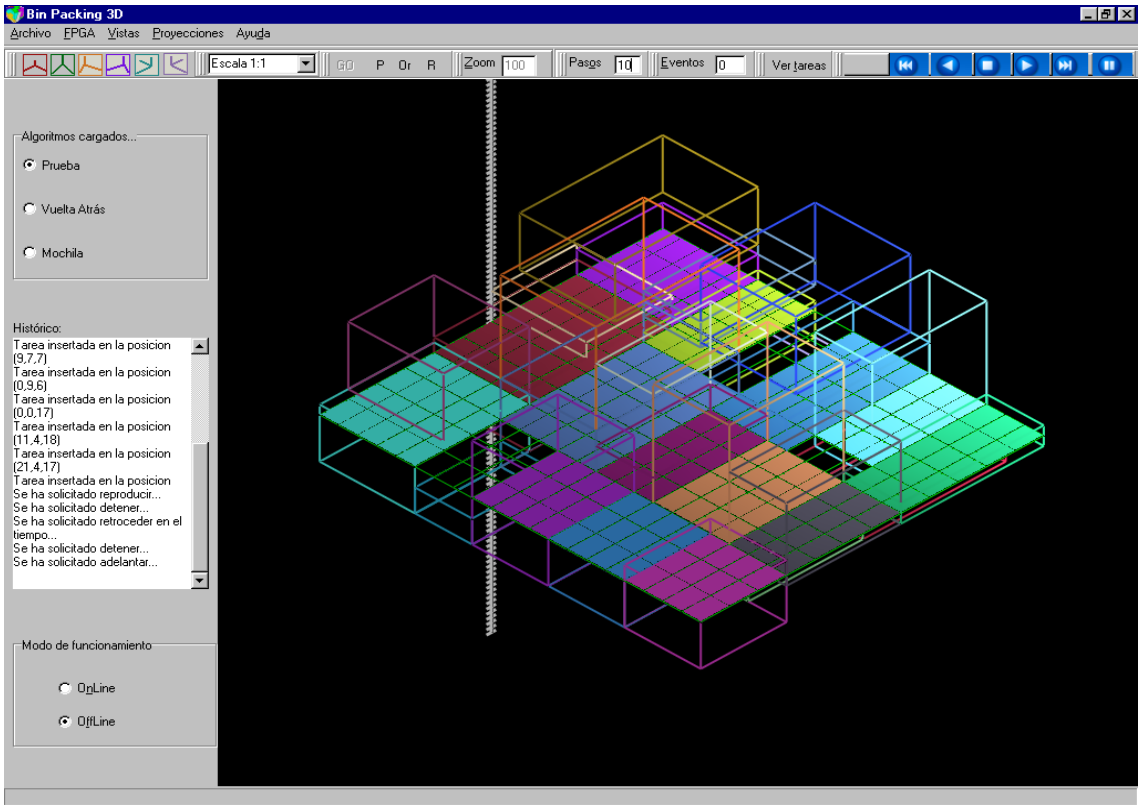
7.1 Controles de reproducción

Los controles de reproducción (azules) reflejan las típicas opciones de video (que esto no lleve a error, nuestro programa no reproduce video), pudiendo escoger entre manejarlos con el ratón o bien mediante teclado. Estas opciones son, de izquierda a derecha:

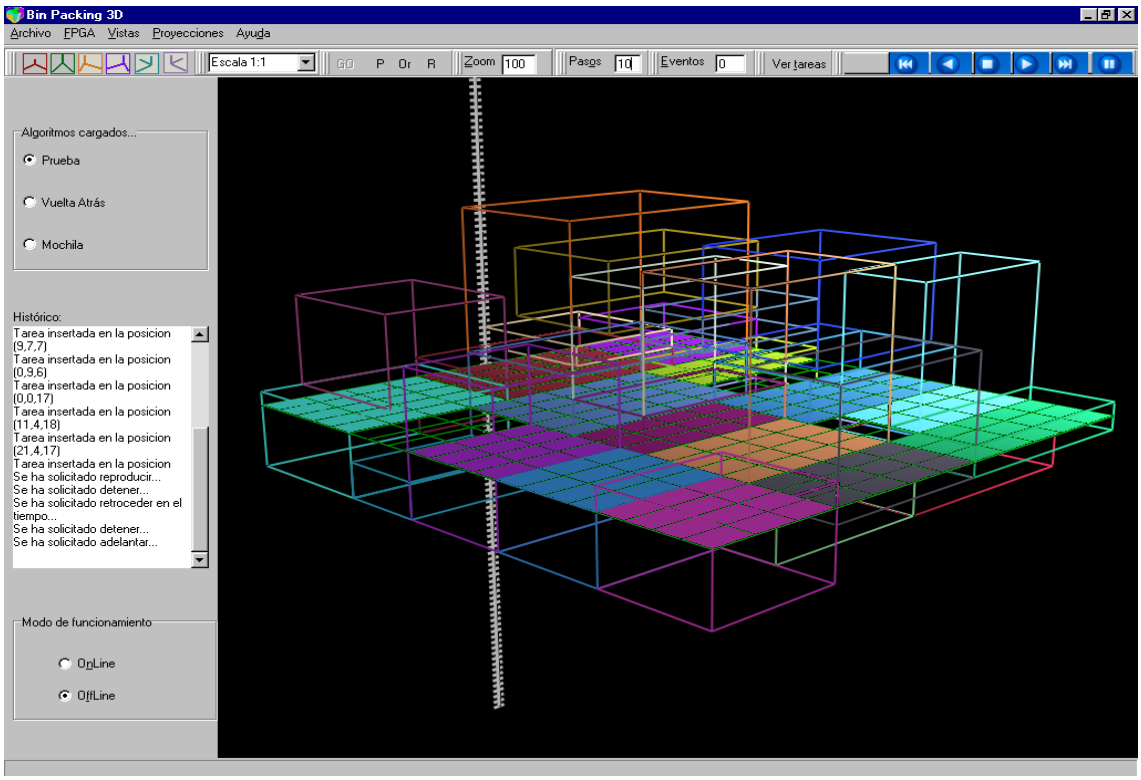
Grabar (ALT+R): comentado en 5. Ejecución.
Retroceder (ALT+W): retrocede la simulación hasta el instante indicado, bien en número de pasos (instante anterior al actual en un número de unidades de tiempo indicadas en Pasos) o en eventos (similar pero se retrocede tantos eventos como indique Eventos).
Retroceder continuo (ALT+G): reproduce la simulación hacia atrás, paso a paso.
Stop (ALT+S): detiene la simulación volviendo al instante inicial.
Avance continuo (ALT+X): reproduce la simulación como sucedería en realidad (PLAY).
Avanzar (ALT+D): avanza la simulación de manera similar a retroceder pero hacia delante, obviamente.
Pausa (ALT+C): Congela la simulación.

7.2 Proyecciones

Con el fin de ofrecer distintas posibilidades a la hora de visualizar, se han creado dos posibles proyecciones de cara a tener la visión que más nos guste, o nos convenga, estas son ortogonal:

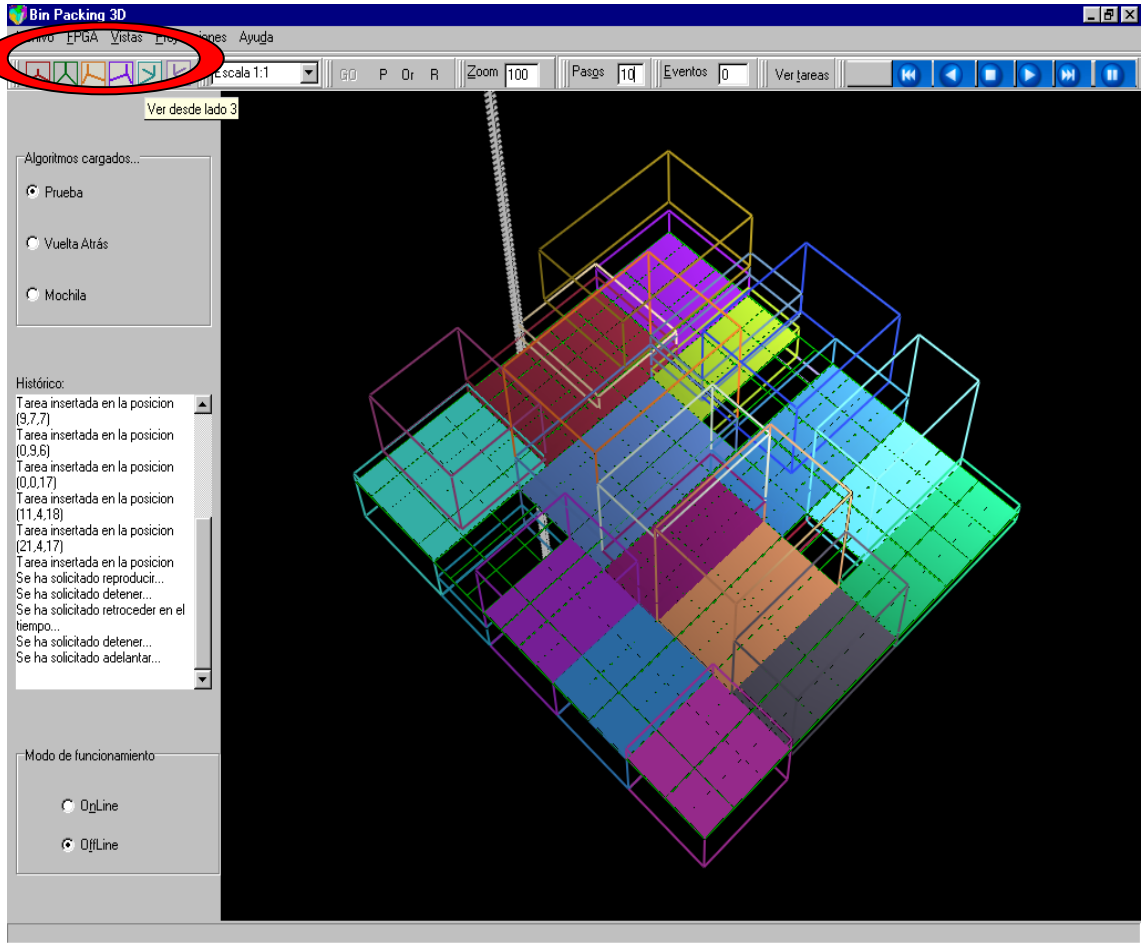


o perspectiva:

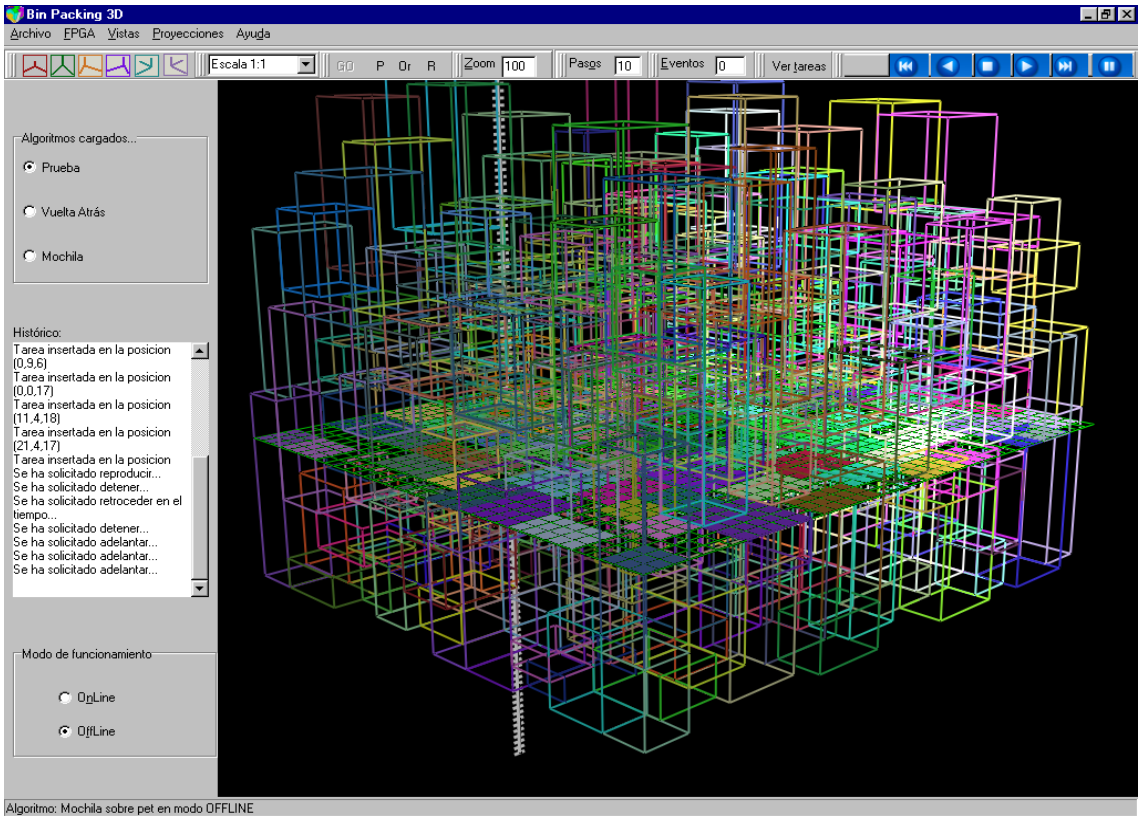


7.3. Vistas

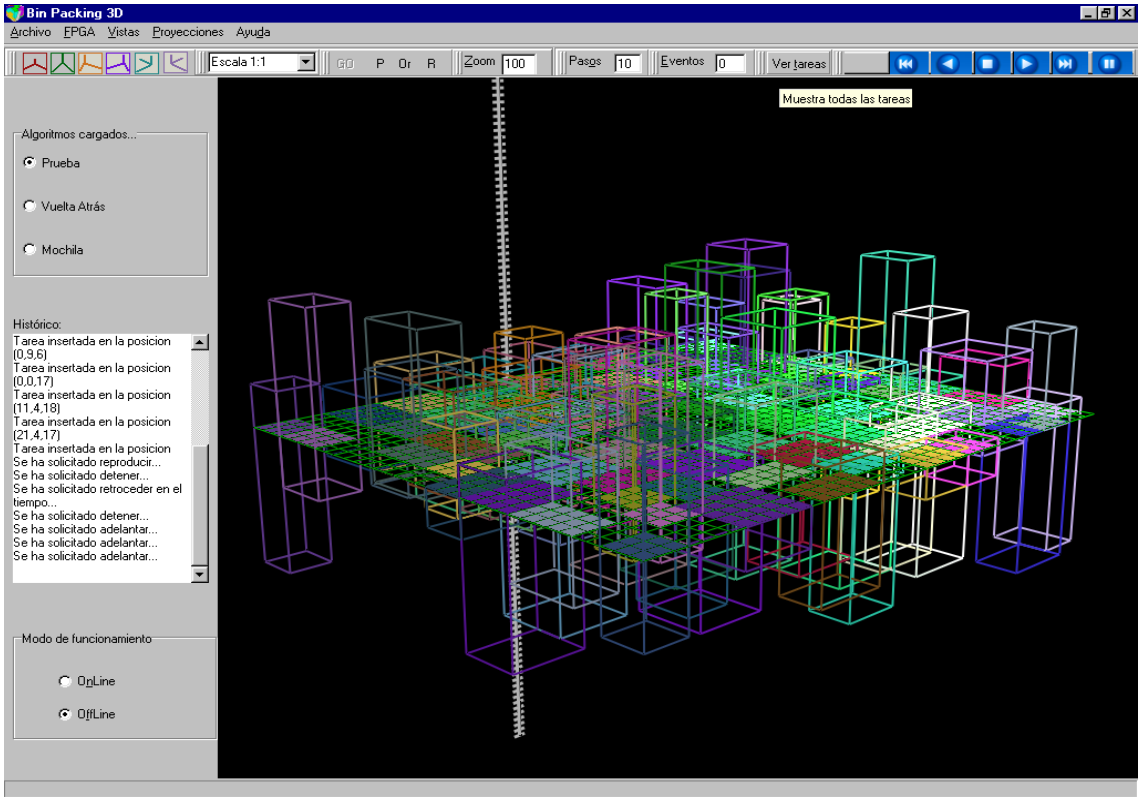
También se ofertan varios puntos de vista para poder acceder mejor al punto exacto que queremos comprobar, esto incluye una rotación para la cual debemos estar colocados en perspectiva. Las distintas vistas se incluyen en el menú Vistas y en los botones de la parte superior izquierda, incluyendo una vista para cada lado, la ya mencionada rotación, y la opción de elegir la vista "normal" (como la anterior) o bien "por encima":



Además, podemos escoger entre ver todas las tareas procesadas o bien únicamente las que en ese momento están en ejecución, para ello usaremos el menú Vistas->Tareas o el botón situado inmediatamente a la izquierda de los controles de reproducción: Ver Tareas:



Todas las tareas

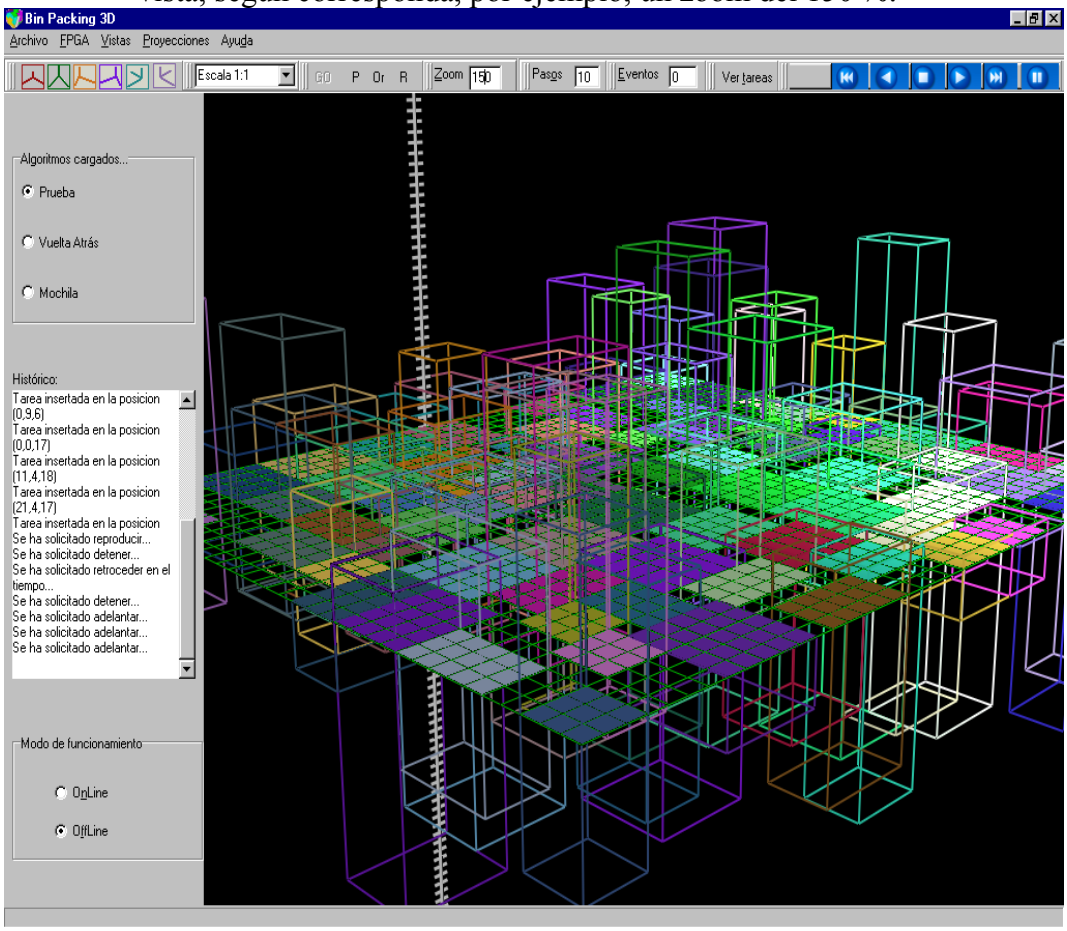


Sólo en ejecución.

7.4. Otras funcionalidades.

7.4.1.El zoom

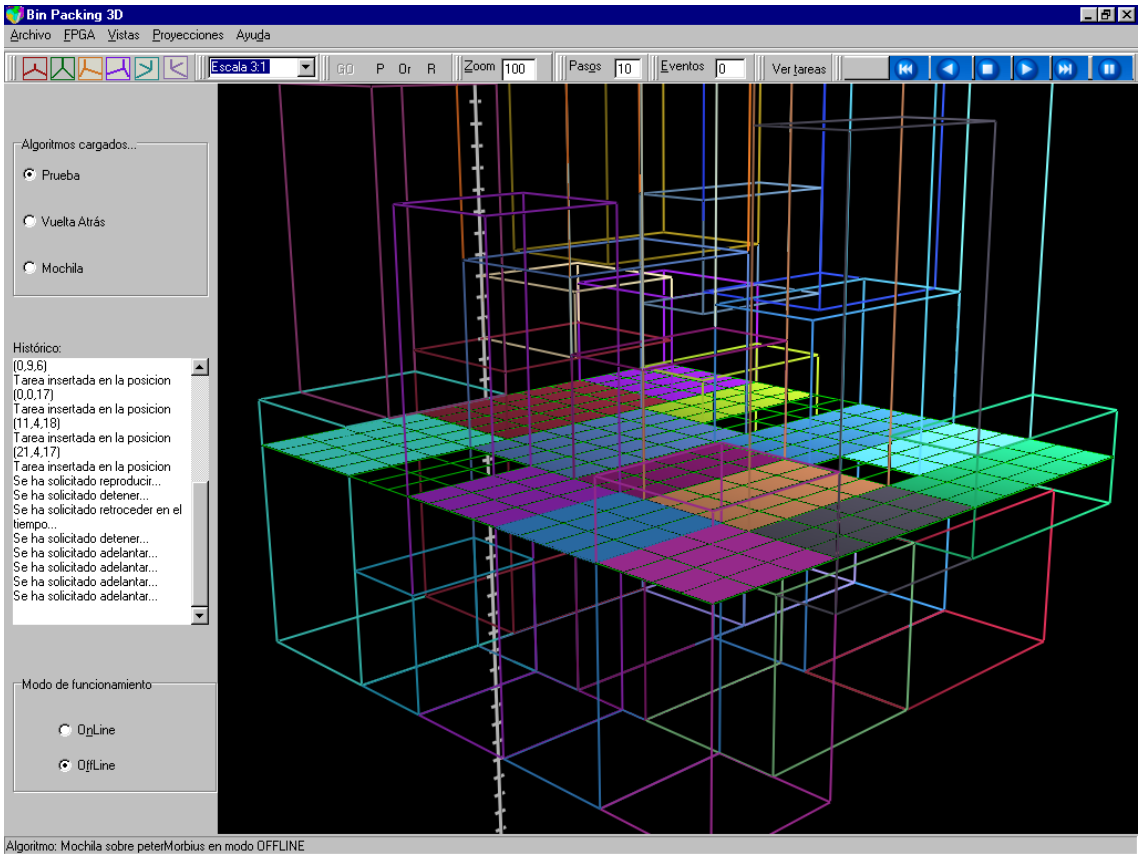
Muestra el tamaño escogido, acercando o alejando el punto de vista, según corresponda, por ejemplo, un zoom del 150 %:



Sólo puede realizarse cuando la proyección está en perspectiva.

7.4.2. La escala

La escala modifica el valor el valor del eje temporal (línea blanca vertical con muescas por cada unidad de tiempo) para tener una mejor referencia de la simulación:



A.2. NOTAS

Tanto la carga de tareas como de FPGAs procesan cualquier fichero, esto conlleva que debemos estar seguros de la corrección de los datos puesto que no se nos avisará de un posible error, simplemente, lo que no se ajuste a la especificación será ignorado.

La funcionalidad FPGA->Tarea->Resetea lista vuelve a colocar las posiciones asignadas de la lista cargada sobre la FPGA a –1, de manera que para volver a procesarla por el mismo u otro algoritmo no haría falta volver a cargarla desde fichero.

A.3. BIBLIOGRAFÍA

"El lenguaje de programación C++"
Bjarne Stroustrup

"Programación con Borlan C++ Builder"
Francisco Charte

"OpenGL programming Guide" (Red Book)
Mason Woo, Jackie Neider, Tom Davis, Dave Sheiner

Apuntes de Informática Gráfica de Puri Arenas y Ana Gil

Articulo:
Fast template placemente for reconfigurable computing System
K. Bazargan, R. Kastner, M. Sarrafzadeh

A.4. AUTORIZACIÓN:

Los alumnos abajo firmantes, autores del proyecto Simulación de algoritmos de binPacking 3-D, Zadith Dianderas La Torre, Fernando López Teso y Javier Rodríguez de Pastors autorizan a la Universidad Complutense de Madrid a difundir y utilizar con fines académicos, no comerciales y mencionando expresamente a dichos autores, esta memoria, así como el código y el propio prototipo desarrollado.

Firmado:

Dianderas La Torre, Zadith

López Teso, Fernando

Rodríguez de Pastors, Javier

A.5. GLOSARIO

Bin Packing:

Técnica de empaquetamiento de varias tareas hardware en una FPGA

FPGA:

Field Programmable Gateway Array, dispositivo digital programable con la capacidad de reconfigurarse en "caliente"

Tarea Hardware:

Módulo a ejecutar en la FPGA

Archivo histórico:

Fichero que contiene una lista de tareas procesada por un determinado algoritmo en una FPGA empleado para realizar la simulación.

GUI:

(Graphics User Interfaz) Interfaz gráfico que comunica al usuario con la aplicación.

Modo OnLine:

Versión de algoritmo en la que las decisiones sobre la colocación de las tareas hardware son tomadas según estas llegan a la FPGA.

Modo OffLine:

Versión de algoritmo en la que se conoce de antemano todas y cada una de las tareas hardware que se van a ejecutar.

Escala:

Tamaño relativo del eje temporal (vertical) con respecto a los otros dos.

Usuario avanzado:

Persona capacitada para manipular el código, quien presumiblemente añadirá nuevos algoritmos.

Altura:

Tiempo que tarda una tarea hardware en ser procesada.